

“十二五”普通高等教育本科国家级规划教材

普 通 高 等 教 育 精 品 教 材

配套用书



21世纪大学本科计算机专业系列教材

吴 英 编著

计算机网络软件编程指导书（第2版）

http://www.tup.com.cn

- 根据教育部“高等学校计算机科学与技术专业规范”组织编写
- 与美国 ACM 和 IEEE CS *Computing Curricula* 最新进展同步
- 国家级精品教材配套用书

清华大学出版社

“十二五”普通高等教育本科国家级规划教材配套用书
21 世纪大学本科计算机专业系列教材

计算机网络软件编程指导书(第 2 版)

吴 英 编著

清华大学出版社
北 京

内容简介

本书根据计算机网络与Internet基本概念、工作原理与实现技术的学习需要，参考国内外知名大学网络课程编程训练以及著名IT企业在员工网络软件编程训练中的相关资料与文献，总结提炼出14个网络软件编程题目，分为3个不同的难度级，力求做到"结合网络课程的教学过程，通过完成实际网络编程课题训练，加深对网络基本原理与实现方法的理解，掌握网络环境中软件编程的基本方法，逐步提高网络软件编程能力"。

本书是"十二五"普通高等教育本科国家级规划教材《计算机网络(第4版)》(主教材)的配套教材，书中第3～16章每章对应一个编程题目。每章包括编程题目的设计目的、相关知识、例题分析和练习题。作者针对不同程度与不同要求的读者，对练习题的选择与进度安排提出了建议。本书可以与主教材配套使用，也可以独立使用。读者可以根据自身的基础与学习要求选择编程题目。完成本书编程题目不需要专门的网络环境与特殊的编程条件。

本书可以作为高等学校计算机专业、软件工程专业、电子信息类专业以及其他相关专业的计算机网络、网络软件编程技术等课程的教材或参考书，也可以作为从事计算机网络应用与信息技术的工程技术人员继续学习和研发工作的参考书。

图书在版编目(CIP)数据

计算机网络软件编程指导书/吴英编著. —2版. —北京：清华大学出版社，2017
(21世纪大学本科计算机专业系列教材)
ISBN 978-7-302-48131-7

Ⅰ. ①计…　Ⅱ. ①吴…　Ⅲ. ①计算机网络－程序设计－高等学校－教材　Ⅳ. ①TP393.09

中国版本图书馆CIP数据核字(2017)第208436号

责任编辑：张瑞庆
封面设计：常雪影
责任校对：李建庄
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：13.75　　**字　　数**：333千字
版　　次：2008年1月第1版　2017年11月第2版　　**印　　次**：2017年11月第1次印刷
印　　数：1～2000
定　　价：29.00元

产品编号：076522-01

前　言

FOREWORD

Internet应用技术、无线网络技术和网络安全技术的研究与发展，使得计算机网络技术进入了一个更高的阶段，正在对社会产生着前所未有的影响。计算机网络已经和电力、电话一样，成为支持现代社会整体运行的基础设施。目前，网络技术发展迅速，应用广泛，知识更新快，产业发展势头强劲，是一个充满活力与机遇的领域。

社会对网络人才的需求十分强烈，但是真正懂得网络技术、具备深入网络协议内部的高层次网络应用系统设计和网络软件编程的软件人才非常缺乏，他们也是社会急需的高级专业人才。作者作为一名青年教师，从个人发展经历中深深地体会到，仅通过课堂听课与课后复习的方法来学习网络技术是不可能达到真正“掌握”网络技术的目的的。本人对网络理论知识的理解与实际动手能力的提高是在网络课程学习的基础上，通过参加科研工作和完成开发任务的过程中“悟”出来的。

作者在带本科生毕业设计的过程中，发现很多计算机专业的本科学生编程能力不是很强，对网络编程也没有入门。本科毕业生在求职过程中反映出的实际动手能力弱的缺陷，与课程教学过程中的硬件实验与软件编程训练严重不足直接相关。为了提高教学质量，提高学生就业的竞争力，必须加强实践环节的训练。目前，网络课程教学急需解决理论与实际的结合，加强学生实际能力的培养。现代的软件都是运行在网络环境中，如果能将两者有机、紧密地结合起来，让学生通过网络软件编程的训练过程来加深对网络理论的理解，同时又能提高学生网络软件编程的能力，作者认为这种训练是十分必要的。

基于这样的认识，作者听取了南开大学网络实验室教师和学生的意见，在总结网络实验室多年的科研工作经验与本科、研究生教学工作实践经验的基础上，根据计算机网络与Internet基本概念、基本工作原理与实现技术学习的需要，参考国内外知名大学网络课程编程训练以及著名IT企业在员工网络软件编程训练中的相关资料与文献，总结提炼出14个网络软件编程题目。软件编程题目的选择考虑不同协议层次的覆盖问题，同时将软件编程题目分为三个难度级，读者可以参考选题指导，根据不同的要求和不同的基础，有选择、循序渐进地完成网络软件编程训练，实现“通过实际编程课题的训练，达到深入理解网络基本工作原理，掌握网络环境中软件编程的基本方法，提高网络软件编程能力”。

本书是“十二五”普通高等教育本科国家级规划教材《计算机网络(第4版)》(主教材)的配套教材。第1章是网络软件编程练习要求与教学指导。第2章是书中编程题目需要用到的Socket编程基础知识。第3～16章每章对应一个编程题目，包括编程训练的设计目的、相关知识、例题分析(含设计要求、关键问题和程序源代码)和练习题。作者针对不同程度与

不同要求的读者,对训练课题的选择与进度安排提出了建议。本书可以与主教材配套使用,也可以独立使用。由读者根据自身的基础与学习要求选择编程题目,循序渐进地学习和独立完成网络软件编程训练。完成本书编程题目不需要专门的网络环境与特殊的编程条件。

本书可以作为高等学校计算机专业、软件工程专业、电子信息类专业以及其他相关专业的学生学习计算机网络、网络软件编程技术等课程的教材或参考书,也可作为从事计算机网络应用与信息技术的工程技术人员继续学习和研发工作的参考书。

作者在本书的编写过程中,得到南开大学计算机与控制工程学院网络与信息安全研究室的教师们的很多支持和指导,特别感谢吴功宜教授、徐敬东教授、张建忠教授的指导和帮助。

限于作者学术水平与经验的不足,错误与不妥之处在所难免,诚恳地希望读者批评指正。

作　者

于南开大学计算机与控制工程学院

2017年9月

目录

CONTENTS

第1章　网络软件编程练习要求与教学指导 …… 1

1.1　网络软件编程能力培养 …… 1
1.2　网络软件编程理论基础 …… 2
1.2.1　网络知识结构 …… 2
1.2.2　编程需掌握的知识 …… 3
1.2.3　教材章节与知识点结构 …… 8
1.3　编程题目的基本内容 …… 9

第2章　Socket 编程基础知识 …… 13

2.1　Socket 编程的基本概念 …… 13
2.1.1　套接字的概念 …… 13
2.1.2　套接字的分类 …… 14
2.2　Winsock 网络编程接口 …… 15
2.2.1　Winsock 的基本概念 …… 16
2.2.2　初始化与卸载 Winsock …… 17
2.2.3　基本 Socket 函数 …… 18
2.2.4　套接字地址结构 …… 23

第3章　Ethernet 帧的封装与解析 …… 25

3.1　设计目的 …… 25
3.2　相关知识 …… 25
3.2.1　数据链路层的概念 …… 25
3.2.2　Ethernet 帧的结构 …… 26
3.3　例题分析 …… 27
3.3.1　设计要求 …… 27
3.3.2　关键问题 …… 28
3.3.3　程序源代码 …… 31
3.4　练习题 …… 34

第4章 Ethernet帧的CRC校验 …… 35

4.1 设计目的 …… 35
4.2 相关知识 …… 35
4.2.1 CRC校验的概念 …… 35
4.2.2 CRC校验的例子 …… 36
4.2.3 CRC校验的硬件实现 …… 37
4.2.4 CRC校验的主要特点 …… 38
4.3 例题分析 …… 38
4.3.1 设计要求 …… 38
4.3.2 关键问题 …… 39
4.3.3 程序源代码 …… 41
4.4 练习题 …… 44

第5章 IP地址的合法性判断 …… 46

5.1 设计目的 …… 46
5.2 相关知识 …… 46
5.2.1 IP地址的基本概念 …… 46
5.2.2 IP地址的分类方法 …… 47
5.2.3 其他IP地址类型 …… 48
5.2.4 IP地址技术发展 …… 49
5.3 例题分析 …… 51
5.3.1 设计要求 …… 51
5.3.2 关键问题 …… 52
5.3.3 程序源代码 …… 54
5.4 练习题 …… 58

第6章 IP数据包的捕获与解析 …… 60

6.1 设计目的 …… 60
6.2 相关知识 …… 60
6.2.1 网络层的基本概念 …… 60
6.2.2 IP数据包的结构 …… 61
6.3 例题分析 …… 64
6.3.1 设计要求 …… 64
6.3.2 关键问题 …… 65
6.3.3 程序源代码 …… 68
6.4 练习题 …… 72

第 7 章 IP 数据包的分片与重组 …… 74

7.1 设计目的 …… 74
7.2 相关知识 …… 74
7.2.1 IP 包分片的概念 …… 74
7.2.2 IP 包分片的相关字段 …… 75
7.3 例题分析 …… 76
7.3.1 设计要求 …… 76
7.3.2 关键问题 …… 77
7.3.3 程序源代码 …… 78
7.4 练习题 …… 83

第 8 章 IPv6 数据包的封装与解析 …… 84

8.1 设计目的 …… 84
8.2 相关知识 …… 84
8.2.1 IPv4 协议的主要缺点 …… 84
8.2.2 IPv6 协议的基本概念 …… 85
8.2.3 IPv6 数据包的结构 …… 86
8.2.4 IPv6 地址结构 …… 88
8.2.5 IPv6 安全功能 …… 90
8.3 例题分析 …… 90
8.3.1 设计要求 …… 90
8.3.2 关键问题 …… 91
8.3.3 程序源代码 …… 93
8.4 练习题 …… 98

第 9 章 发现网络中的活动主机 …… 100

9.1 设计目的 …… 100
9.2 相关知识 …… 100
9.2.1 ICMP 协议的基本概念 …… 100
9.2.2 ICMP 数据包的类型 …… 101
9.2.3 ICMP 数据包的结构 …… 102
9.2.4 ICMP 回送请求与应答 …… 103
9.3 例题分析 …… 104
9.3.1 设计要求 …… 104
9.3.2 关键问题 …… 104
9.3.3 程序源代码 …… 107
9.4 练习题 …… 112

第10章　发现服务器开启的TCP端口 …… 113

10.1　设计目的 …… 113
10.2　相关知识 …… 113
10.2.1　传输层的基本概念 …… 113
10.2.2　端口号的分配 …… 114
10.3　例题分析 …… 116
10.3.1　设计要求 …… 116
10.3.2　关键问题 …… 116
10.3.3　程序源代码 …… 118
10.4　练习题 …… 120

第11章　TCP数据包的封装与发送 …… 121

11.1　设计目的 …… 121
11.2　相关知识 …… 121
11.2.1　TCP协议的基本概念 …… 121
11.2.2　TCP数据包的结构 …… 122
11.3　例题分析 …… 124
11.3.1　设计要求 …… 124
11.3.2　关键问题 …… 125
11.3.3　程序源代码 …… 126
11.4　练习题 …… 131

第12章　基于TCP的客户机/服务器程序 …… 132

12.1　设计目的 …… 132
12.2　相关知识 …… 132
12.2.1　TCP协议的主要特点 …… 132
12.2.2　客户机/服务器编程 …… 134
12.3　例题分析 …… 135
12.3.1　设计要求 …… 135
12.3.2　关键问题 …… 136
12.3.3　程序源代码 …… 139
12.4　练习题 …… 143

第13章　基于UDP的客户机/服务器程序 …… 144

13.1　设计目的 …… 144
13.2　相关知识 …… 144
13.2.1　UDP协议的基本概念 …… 144
13.2.2　UDP数据包的结构 …… 145

13.2.3　基于 UDP 的客户机/服务器编程 …………………………………… 146
13.3　例题分析………………………………………………………………… 147
13.3.1　设计要求……………………………………………………………… 147
13.3.2　关键问题……………………………………………………………… 148
13.3.3　程序源代码…………………………………………………………… 150
13.4　练习题…………………………………………………………………… 154

第 14 章　FTP 客户机程序设计 ……………………………………………… 155

14.1　设计目的………………………………………………………………… 155
14.2　相关知识………………………………………………………………… 155
14.2.1　应用层的基本概念…………………………………………………… 155
14.2.2　FTP 服务的基本概念 ……………………………………………… 156
14.2.3　FTP 服务的工作原理 ……………………………………………… 157
14.2.4　FTP 命令与应答 …………………………………………………… 158
14.3　例题分析………………………………………………………………… 160
14.3.1　设计要求……………………………………………………………… 160
14.3.2　关键问题……………………………………………………………… 161
14.3.3　程序源代码…………………………………………………………… 163
14.4　练习题…………………………………………………………………… 172

第 15 章　POP 客户机程序设计 ……………………………………………… 174

15.1　设计目的………………………………………………………………… 174
15.2　相关知识………………………………………………………………… 174
15.2.1　电子邮件的基本概念………………………………………………… 174
15.2.2　邮件服务的工作原理………………………………………………… 175
15.2.3　邮件地址与邮件格式………………………………………………… 176
15.2.4　POP 命令与应答 …………………………………………………… 178
15.3　例题分析………………………………………………………………… 179
15.3.1　设计要求……………………………………………………………… 179
15.3.2　关键问题……………………………………………………………… 180
15.3.3　程序源代码…………………………………………………………… 182
15.4　练习题…………………………………………………………………… 189

第 16 章　包过滤防火墙程序设计 …………………………………………… 191

16.1　设计目的………………………………………………………………… 191
16.2　相关知识………………………………………………………………… 191
16.2.1　网络安全的重要性…………………………………………………… 191
16.2.2　防火墙的基本概念…………………………………………………… 192
16.2.3　防火墙的分类方法…………………………………………………… 193

16.2.4 防火墙系统结构…… 194

16.3 例题分析…… 196

16.3.1 设计要求…… 196

16.3.2 关键问题…… 196

16.3.3 程序源代码…… 198

16.4 练习题…… 204

附录 RFC文档 …… 205

参考文献 …… 207

第1章 网络软件编程练习要求与教学指导

1.1 网络软件编程能力培养

1. 社会对网络软件编程人才的需求

计算机网络是计算机技术与通信技术相互渗透、密切结合而形成的一门交叉科学，同时也正在与其他的专业相结合，促进了相关交叉学科的发展。计算机网络教育正在由开始的普及阶段，进一步地向“扁平化”和“深层次”方向发展。“扁平化”表现在网络课程的教学正在从计算机专业向相关专业发展；“深层次”表现在社会急需大量网络软件高级专门人才。计算机网络是当今计算机科学与技术学科中发展最快的技术之一，也是计算机应用中一个空前活跃的领域。无论是工科、理科，甚至是文科(包括财经、政法、艺术类)，例如计算机、软件工程、网络工程、信息安全、大众传媒、电子商务、物流、平面设计等专业，很多课程的学习都是建立在学生掌握计算机网络知识的基础上。

计算机网络又是一个技术性很强的课程，完整的网络技术训练主要包括基本组网技术的训练和网络环境软件编程技术的训练。计算机网络已经成为软件编程的基本环境。计算机、网络工程与信息安全专业的学生，都需要具备在网络环境中完成软件编程的能力。社会对网络软件编程人才的需求日趋旺盛。

从20世纪90年代开始，我国也和一些发达国家一样，迅速地向信息化社会迈进。社会信息化初期的主要任务是建设覆盖全社会的网络基础设施。我国信息技术与产业的发展，需要大量从事网络应用系统设计、系统集成、软件工程、电信技术、信息服务与各类信息系统管理的专业技术人员，以及网络与信息系统的使用和维护人员。但是，投入大量资金、铺设大批光纤、建设网络系统与构建信息高速公路不是目的，这只是信息化社会发展过程中必须经过的第一个阶段，它只能解决信息化社会的“路”的问题。社会信息化的最终目的是通过社会的信息化去推动经济发展，协调解决好“路”“车”和“货”的关系。这些都离不开网络软件编程技术、人才与产业的支持。

随着我国信息化进程的发展，社会对人才的需求从信息高速公路设计和建设人才的初级阶段，逐步向信息系统、信息资源与服务系统建设，以及信息系统安全、高效运行管理的网络软件人才的高级阶段发展。

2. 网络软件编程对网络理论课程学习的促进

21世纪的一个重要特征是数字化、网络化与信息化，它的基础是支持全社会强大的计

算机网络。随着计算机网络技术的广泛应用,计算机网络知识的学习变得非常重要,各种类型的学校以及各个层次和各个专业的学生都需要学习计算机网络课程。但是,仅通过书本和课堂学习的网络知识是初步的,根本谈不上“掌握”。

通过对已经毕业的学生进行追踪调查发现:学生对计算机网络理论的真正理解与实际工作能力的培养,是在参加科研工作和完成开发任务的过程中“悟”出来和“干”出来的。他们在学习计算机网络课程时,通常只是开始了解计算机网络的一般性知识,以及对网络实现方法有一个比较模糊的了解。

如果只是在课堂上讲授网络协议与实现方法,学生会感到很陌生和枯燥,经常提不起兴趣,觉得不好理解,无法掌握。教师在计算机网络课程的考试命题时,只能采取问答题与选择题等几种简单形式。如果将网络协议实现中的重要问题与普遍使用的方法变成软件编程习题,让学生在进行网络理论课程学习的同时,循序渐进地通过完成编程练习来将理论与实际相结合。这样,一方面有助于消除学生对网络协议的神秘感,激发学生对学习的兴趣与热情,调动学生的主动性和积极性;另一方面能帮助学生在完成练习的过程中,逐步理解网络理论知识的精髓,提高知识深度与学习质量。同时,学生能逐步掌握网络软件编程的基本方法与技巧,提高学生就业的竞争力。

尽管计算机网络与软件编程课程同属于计算机专业的必修课程,网络课程的教学内容中也不可能离开软件实现技术,并且现代的软件多数是运行在网络环境中,但是在实际教学过程中还没有很好地将二者有机、紧密地结合起来。通过网络软件编程的练习过程,学生可以加深对网络理论的理解,同时又能提高网络软件编程能力。目前,计算机网络教学急需解决好理论与实际的结合,加强学生实际工作能力的培养。网络软件编程练习对于深入理解网络工作原理与实现技术是至关重要的手段之一。

为了适应学生对计算机网络技术学习的需要,本书结合作者多年科研与教学工作的实践经验,参考国内外知名大学的网络课程编程练习以及著名 IT 企业在员工网络软件编程练习中的相关资料与文献,配合《计算机网络(第 4 版)》教材的教学过程,总结提炼出 14 个网络软件编程题目,按照数据链路层、网络层、传输层、应用层与网络安全的结构,将这 14 个题目划分为 5 个部分、3 个不同的难度级,由教师根据教学的需要和进度,或读者根据自身的基础与学习要求来选择编程题目,循序渐进地学习和独立完成网络软件编程练习。

本书安排的编程练习不需要任何特殊的硬件环境和编程语言的支持,学习过“高级语言程序设计”的学生都可以按照教学指导,根据网络课程学习的要求或教师的安排,循序渐进地完成编程练习。本书中的各个题目之间没有前后顺序的约束关系,读者可以根据自己的基础与兴趣独立地选择练习内容。

本书按照《计算机网络(第 4 版)》的配套教材的思路编写,但是也可以独立于该教材单独使用。

1.2 网络软件编程理论基础

1.2.1 网络知识结构

按照《计算机网络(第 4 版)》教材的组织思路,将计算机网络技术所涉及的问题分为计

算机网络概论、物理层、数据链路层、介质访问控制子层、网络层、传输层、应用层、网络安全与网络技术发展，对广域网、局域网与城域网、网络互联、分布式进程通信、Internet 应用、网络安全等技术进行系统的介绍。这种结构的特点是：采用层次结构的设计思想，但是并不拘泥于传统 OSI 参考模型的结构，实际的层次结构采用 TCP/IP 参考模型与协议集。图 1-1 给出了网络教材的知识点结构。

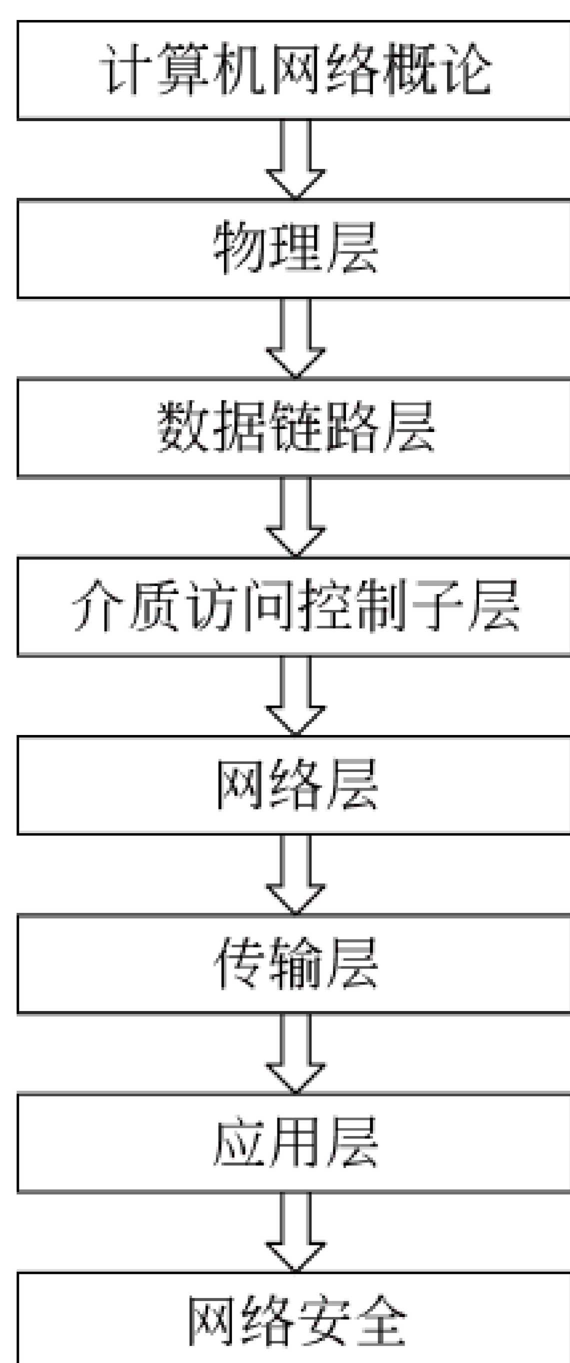


图 1-1　网络教材的知识点结构

作为网络软件编程学习的基础，希望读者具备解答以下问题的基础知识。

- 什么是计算机网络？
- 什么是处理计算机网络问题的基本方法？
- 如何实现广域网中计算机之间的通信？
- 如何保证广域网中计算机通信的可靠性？
- 如何实现局域网与城域网中计算机之间的通信？
- 如何实现网络互联？
- 如何实现网络中计算机之间的分布式进程通信？
- 如何设计和实现 Internet 服务功能？
- 如何保证网络安全？

1.2.2　编程需掌握的知识

以上讨论了计算机网络的知识点结构，这些知识点在不同的教材中都会涉及。对于完成网络软件编程需要掌握的网络知识，我们按以上划分的层次进行简单的回顾，帮助读者顺利进入网络软件编程的学习。

1. 计算机网络概论

读者需要掌握计算机网络形成与发展、计算机网络技术发展主线、计算机网络定义与分类、计算机网络组成与结构、计算机网络拓扑结构、分组交换技术，以及网络体系结构与网络协议等基本问题。下面以典型的计算机网络与数据通信服务为例，对网络在政府部门、企业信息管理与个人信息服务中的各种应用，以及网络应用带来的社会问题进行系统的介绍。图 1-2 给出了计算机网络概论的知识点结构。

本章的学习要求如下。

- 了解：计算机网络的形成与发展过程。
- 了解：计算机网络技术发展的三条主线。
- 掌握：计算机网络的定义与主要类型。
- 掌握：计算机网络的组成与结构。
- 掌握：计算机网络的拓扑结构与分类。
- 了解：分组交换技术的概念。
- 掌握：网络体系结构与网络协议的概念。

2. 物理层

数据通信技术是计算机网络技术发展的基础。读者需要掌握物理层与物理层协议的概念、数据通信的基本概念、频带传输技术、基带传输技术、多路复用技术、SONET标准与SDH体系,以及接入网技术等几个基本问题。针对计算机网络底层的数据通信问题,对物理层的传输介质、编码方式与多路复用技术,以及利用上述技术的接入网进行系统的介绍。图1-3给出了物理层的知识点结构。

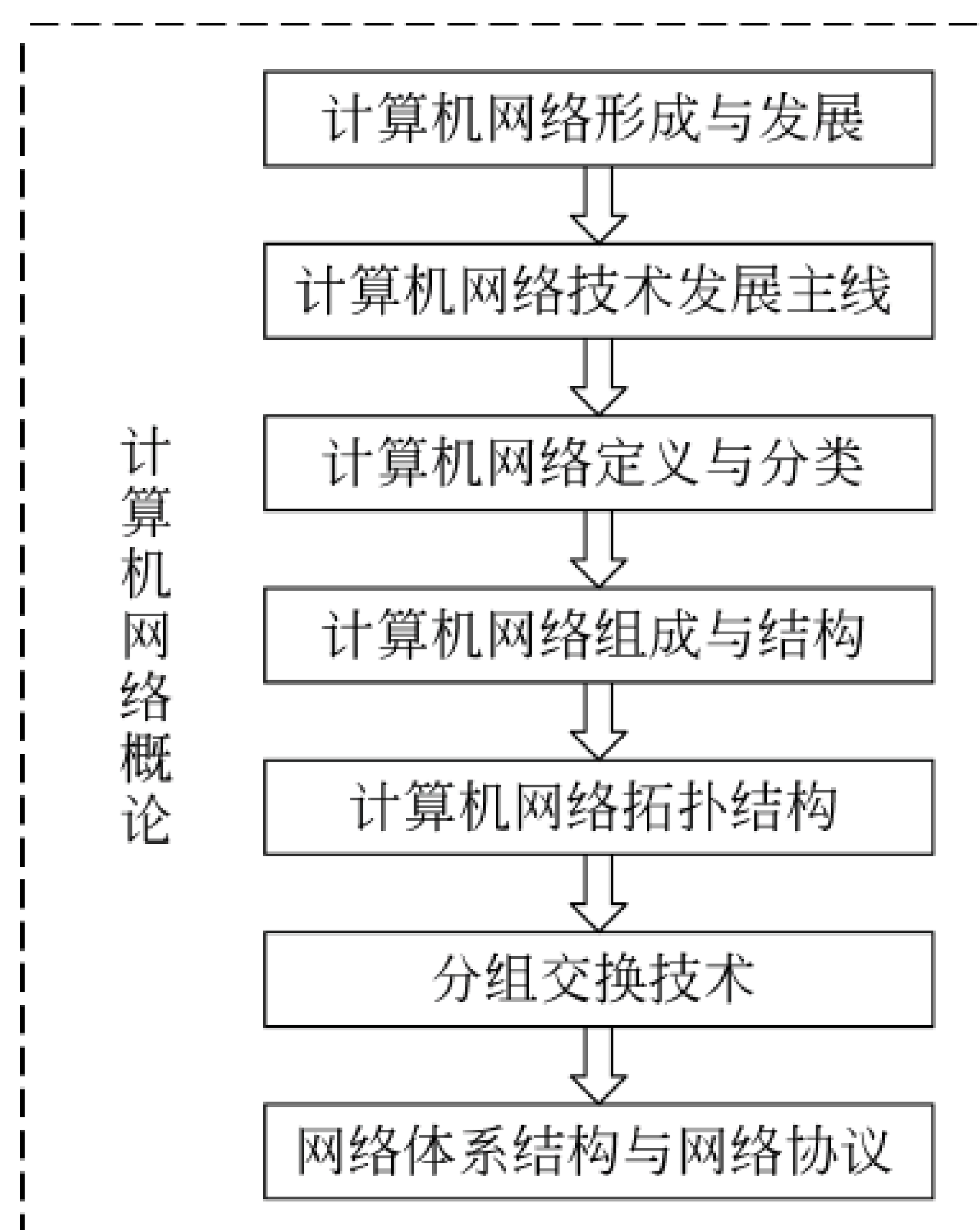


图1-2 计算机网络概论的知识点结构

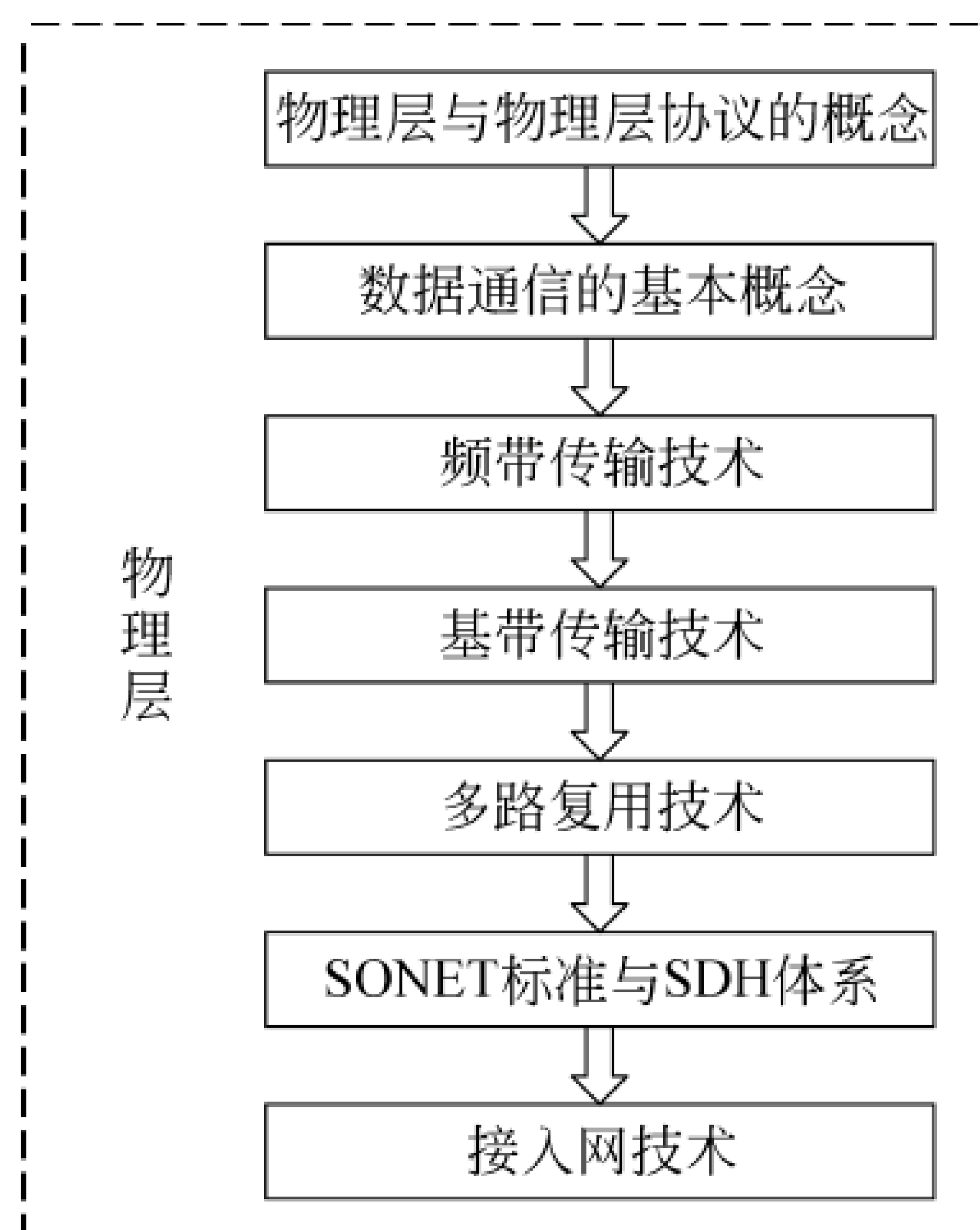


图1-3 物理层的知识点结构

本章的学习要求如下。

- 掌握:物理层与物理层协议的概念。
- 掌握:数据通信的基本概念。
- 了解:频带传输技术的概念。
- 掌握:基带传输技术的概念。
- 了解:多路复用技术的概念。
- 了解:SONET标准与SDH体系。
- 掌握:接入网技术的主要类型。

3. 数据链路层

在掌握基于点-点通信线路的物理层协议与标准的基础上,进一步掌握差错产生的原因与差错控制方法、基于点-点通信线路的数据链路层的基本概念、服务功能与标准,以及典型的数据链路层协议。图1-4给出了数据链路层的知识点结构。

本章的学习要求如下。

- 了解:数据传输过程中差错产生的原因与性质。
- 掌握:误码率的定义与差错控制方法。
- 掌握:数据链路层的基本概念。
- 了解:面向字符型数据链路层协议实例(BSC)。
- 掌握:面向比特型数据链路层协议实例(HDLC)。
- 掌握:Internet中的数据链路层协议(PPP)。

4. 介质访问控制子层

要求读者在介质访问控制子层关键技术与标准的基础上，掌握共享介质局域网、交换局域网与高速局域网的工作原理与组网方法，掌握 Ethernet、高速局域网、交换局域网、无线局域网与虚拟局域网的工作原理，掌握局域网互联的基本概念和网桥的工作原理，初步具备局域网组网的基础知识与能力。图 1-5 给出了介质访问控制子层的知识点结构。

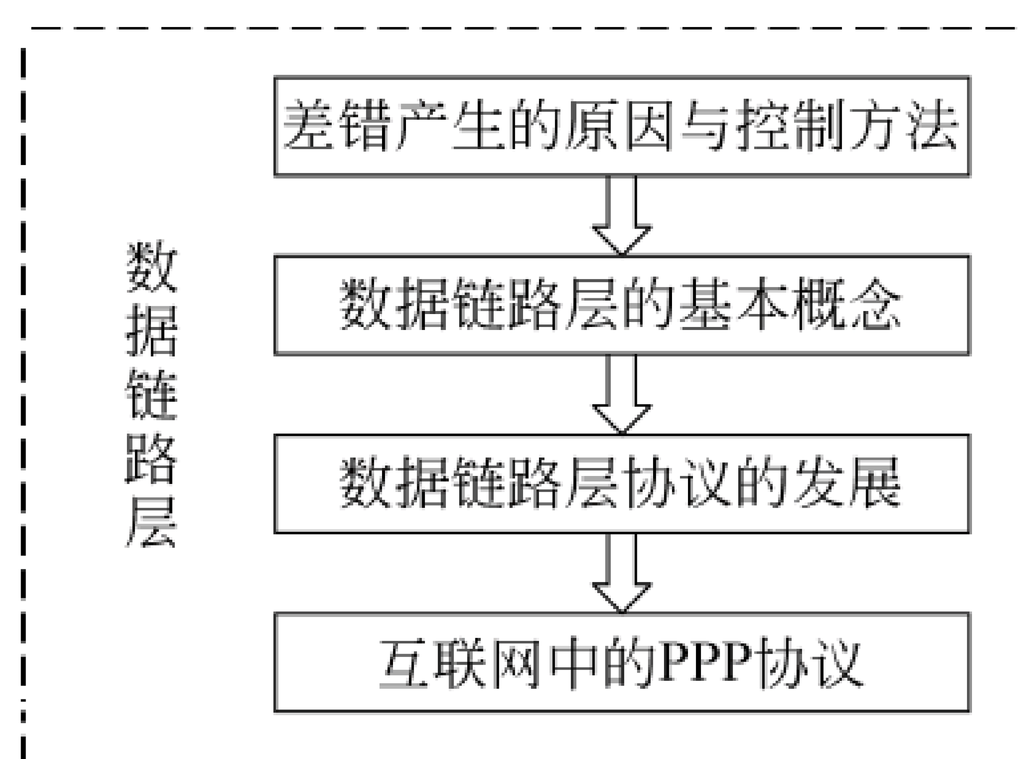


图 1-4　数据链路层的知识点结构

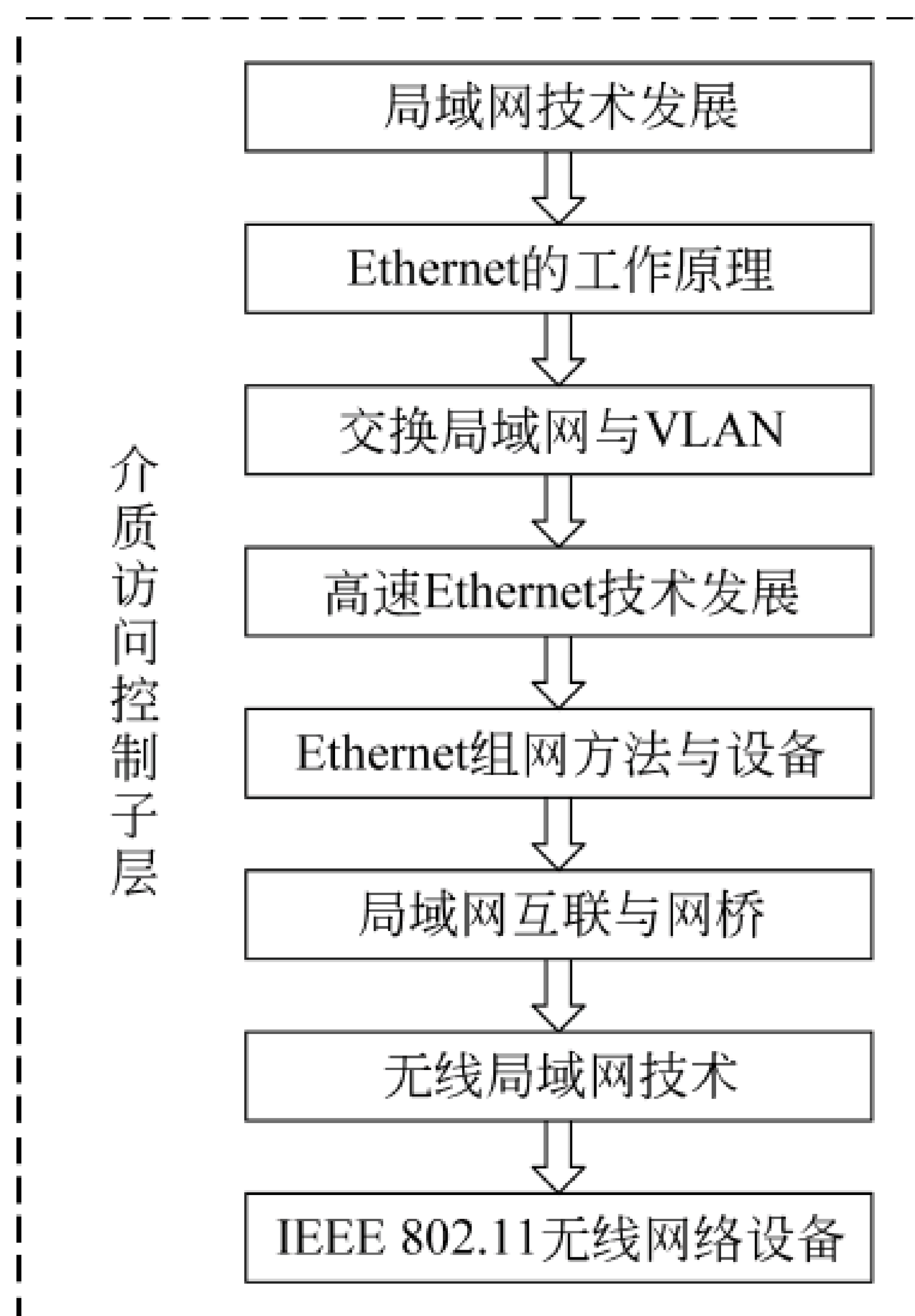


图 1-5　介质访问控制子层的知识点结构

本章的学习要求如下。

- 了解：局域网与城域网的主要技术特点。
- 了解：局域网拓扑结构的类型与特点。
- 了解：IEEE 802 参考模型与介质访问控制子层协议的基本概念。
- 掌握：Ethernet 的工作原理。
- 掌握：高速局域网、交换局域网与虚拟局域网的工作原理与技术发展。
- 了解：无线局域网的工作原理与技术发展。
- 掌握：网桥的基本工作原理。

5. 网络层

要求读者掌握网络层的基本概念、网络层服务功能、IP 地址、路由选择算法与协议、流量控制算法、IP 协议，以及网络互联的概念与方法、路由器工作原理与设计方法，为进一步研究 Internet 实现技术打下坚实的基础。图 1-6 给出了网络层的知识点结构。

本章的学习要求如下。

- 了解：网络层与网络互联的基本概念。
- 掌握：IP 协议的特点与基本内容。
- 了解：IPv4 协议的基本内容。
- 掌握：IP 地址及地址处理方法。
- 了解：IPv4 地址与改进技术。
- 掌握：地址解析协议(ARP)的基本概念与实现方法。

• 掌握：IP分组的转发与路由选择的概念。
• 掌握：Internet路由选择协议的概念。
• 掌握：路由器与第三层交换的基本工作原理。
• 理解：Internet控制报文协议(ICMP)与Internet组管理协议(IGMP)。
• 掌握：IPv6协议的主要技术特点。

6. 传输层

计算机网络的本质活动是实现分布在不同地理位置的主机之间的进程通信，以实现应用层的各种网络服务功能。传输层的主要作用是实现分布式进程通信，因此它是整个协议结构的核心。要求读者掌握分布式进程通信的基本概念、传输层的基本功能，以及实现这些服务的TCP与UDP协议的基本内容，为读者进一步研究应用层协议打下基础。图1-7给出了传输层的知识点结构。

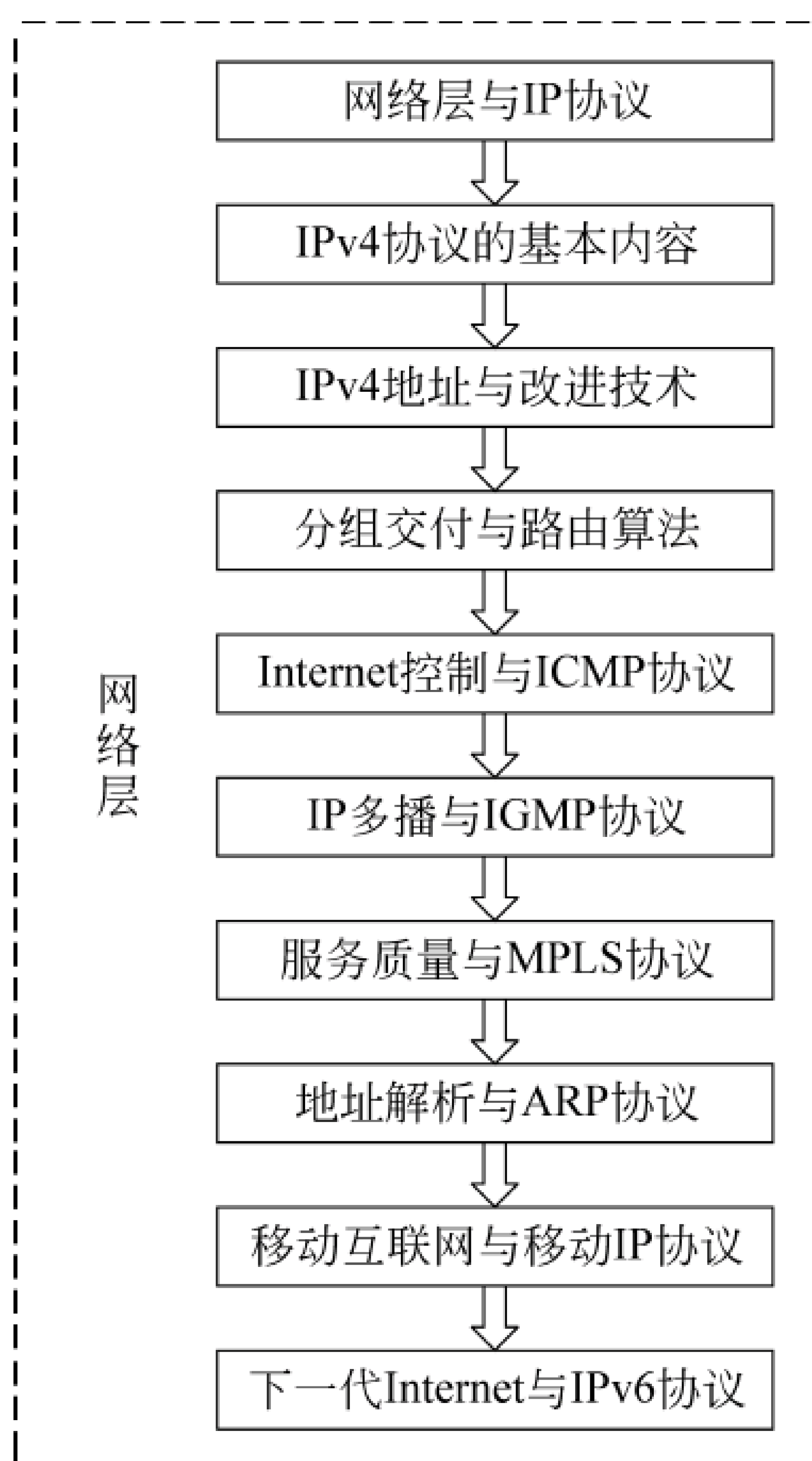


图1-6　网络层的知识点结构

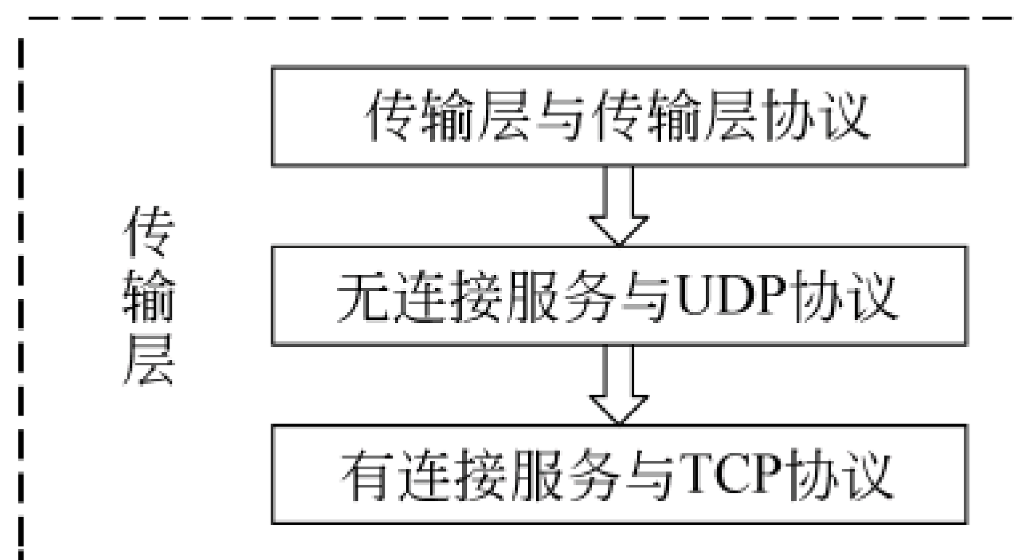


图1-7　传输层的知识点结构

本章的学习要求如下。

• 了解：网络环境中分布式进程通信的基本概念。
• 掌握：进程通信中的客户机/服务器(Client/Server)模式。
• 掌握：传输层的基本功能与服务质量(QoS)的基本概念。
• 掌握：用户数据报协议(UDP)的基本内容。
• 掌握：传输控制协议(TCP)的基本内容。

7. 应用层

要求读者掌握域名系统(DNS)、文件传送协议(FTP)、电子邮件(E-mail)、WWW服务等基本应用层服务的工作原理与协议，深入理解网络协议的层次结构、客户机/服务器的交

互过程、协议与协议动作、协议数据单元与实现技术，为进一步学习 Internet 增值服务的软件系统设计与编程技术打下坚实的基础。图 1-8 给出了应用层的知识点结构。

本章的学习要求如下。

- 了解：TCP/IP 协议族与应用层协议之间的关系。
- 掌握：域名系统的基本工作原理。
- 掌握：电子邮件的基本工作原理。
- 掌握：FTP 服务的基本工作原理。
- 掌握：Web 服务的基本工作原理。
- 掌握：应用层协议的分析方法。

8. 网络安全

要求读者掌握网络安全的基本概念、密码体制的基本概念、主要的网络安全协议，以及防火墙技术、入侵检测技术、网络文件的备份与恢复、网络防病毒技术与网络管理技术。图 1-9 给出了网络安全的知识点结构。

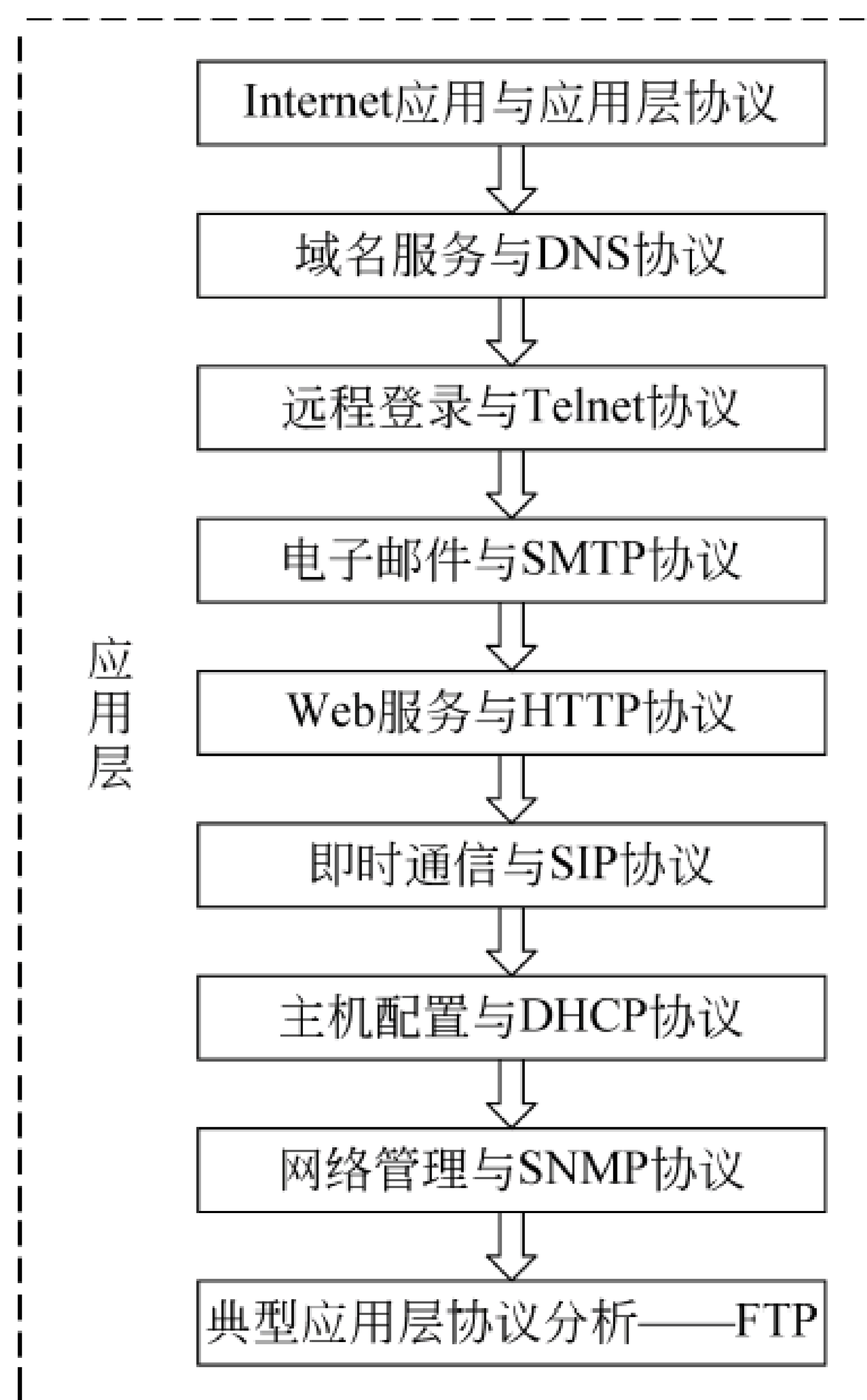


图 1-8　应用层的知识点结构

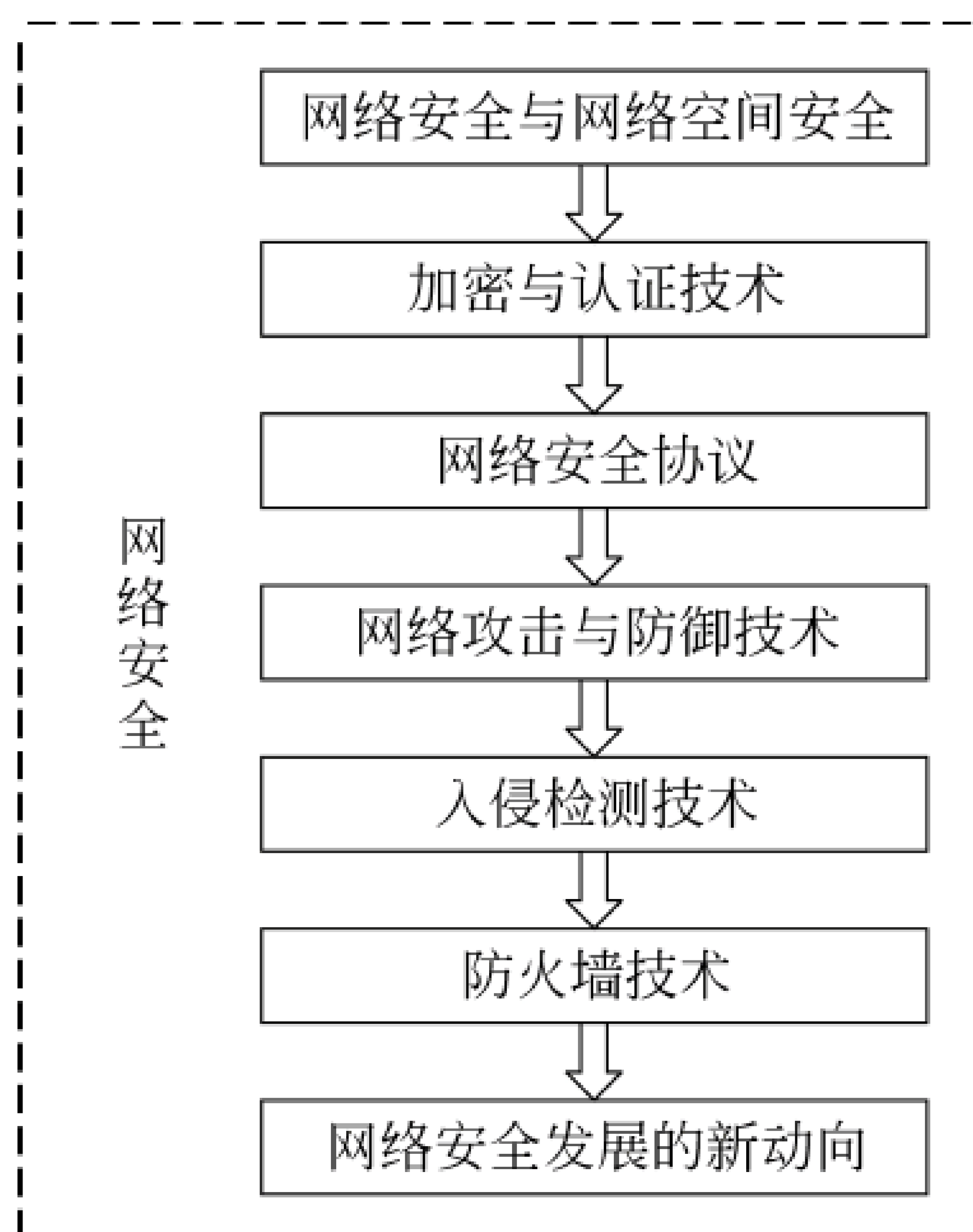


图 1-9　网络安全的知识点结构

本章的学习要求如下。

- 了解：网络安全与网络空间安全的重要性。
- 掌握：网络安全策略制定的方法与基本内容。
- 掌握：密码体制的基本概念及应用。
- 掌握：防火墙的基本概念、工作原理与结构。
- 掌握：网络攻击与防御及入侵检测的基本概念与方法。
- 掌握：网络文件备份与恢复的基本方法。
- 理解：网络病毒防治的基本概念和方法。
- 掌握：网络管理的基本概念、协议与方法。

1.2.3 教材章节与知识点结构

计算机网络课程的教学内容主要包括网络基本概念、网络体系结构,物理层、数据链路层、介质访问控制子层、网络层、传输层与应用层协议,以及网络安全与网络管理技术,等等。计算机网络教学需要学生对广域网、局域网与城域网技术以及网络互联、分布式进程通信、Internet 应用、网络安全技术进行系统的学习。图 1-10 给出了《计算机网络(第 4 版)》教材章节与知识点结构的关系。

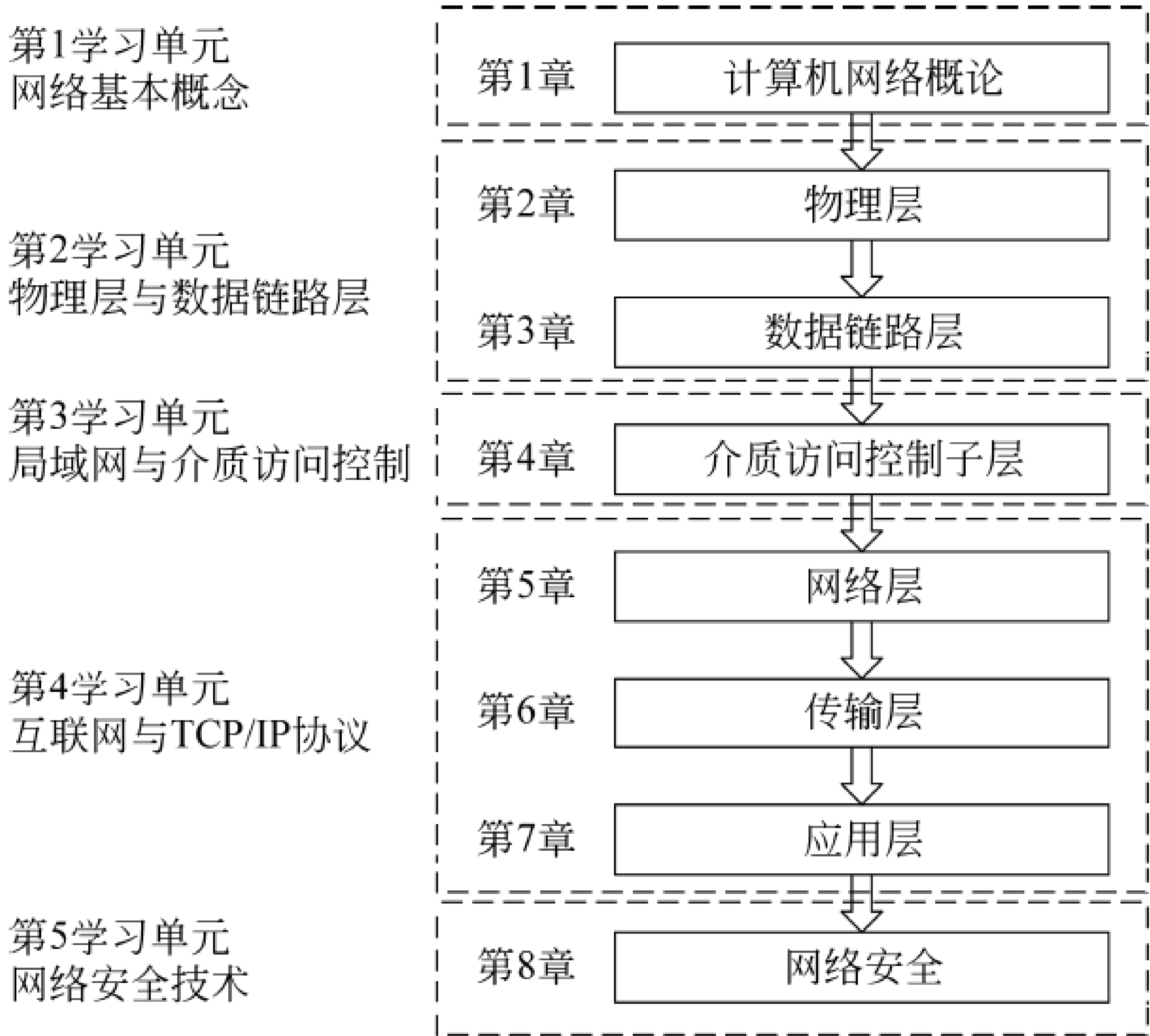


图 1-10　教材章节与知识点结构的关系

从教学内容上来看,全书 8 章可以分成 5 个学习单元。

1. 第 1 学习单元

第 1 个学习单元由第 1 章组成。本单元介绍了计算机网络的基本概念、发展与应用,以及网络体系结构与网络协议的基础知识。

2. 第 2 学习单元

第 2 个学习单元由第 2 章、第 3 章组成。本单元在数据通信基础知识与概念的基础上,对广域网的物理层、数据链路层的基本概念与协议进行了系统的讨论。

3. 第 3 学习单元

第 3 个学习单元由第 4 章组成。本单元在介质访问控制方法的基础上,对局域网、城域网技术发展及应用进行了讨论,同时介绍了交换局域网、虚拟局域网、无线局域网技术与基本组网方法。

4. 第 4 学习单元

第 4 个学习单元由第 5 章、第 6 章、第 7 章组成,对 TCP/IP 协议体系中的网络层、传输层与应用层进行了系统的讨论。本单元对网络互联、分布式进程通信、客户机/服务器模型进行了深入分析,并以典型应用层协议的分析为例,对网络服务的基本概念、协议与协议动作、协议数据单元等基本问题进行了总结,以帮助读者能够将前后学习的知识融会贯通,加深对网络工作原理与实现技术的理解。

5. 第5学习单元

第5个学习单元由第8章组成。本单元介绍了网络安全技术研究的基本问题、网络安全策略设计，加密与认证、网络安全协议、入侵检测、防火墙、网络防病毒、网络文件备份与恢复技术，以及网络管理技术。

编程练习的安排是结合以上教学内容编写的。

1.3 编程题目的基本内容

1. Ethernet帧的封装与解析

帧是在数据链路层中进行数据传输的基本单位。熟悉帧结构对于理解网络协议的概念、网络层次结构、协议执行过程以及网络问题处理的一般方法，具有重要的意义。

本次练习的目的是：

(1) 掌握Ethernet帧的各个字段的含义与用途。

(2) 掌握Ethernet帧结构解析软件设计与编程方法。

2. Ethernet帧的CRC校验

帧的CRC校验是数据链路层提供的差错控制方法。熟悉Ethernet帧的CRC校验过程，对于理解数据链路层的设计思路与工作原理有很大的帮助。

本次练习的目的是：

(1) 掌握Ethernet帧的CRC校验的计算过程。

(2) 理解数据链路层协议的设计思想、工作原理与编程方法。

3. IP地址的合法性判断

IP地址是TCP/IP协议在网络层使用的地址，用来唯一地标识一台连入Internet的主机。熟悉IP地址对于理解网络协议的概念、层次结构与执行过程有很大的帮助。

本次练习的目的是：

(1) 掌握IPv4地址的基本结构与分类方法。

(2) 理解网络层协议的设计思想与工作原理。

4. IP数据包的捕获与解析

IP数据包是在网络层中进行数据传输的基本单位。熟悉IP数据包结构对于理解网络协议的概念、网络层次结构、协议执行过程以及网络问题处理的一般方法，具有重要的意义。

本次练习的目的是：

(1) 掌握IP数据包头部中各个字段的含义与用途。

(2) 掌握通过网卡截获经过的IP数据包的基本方法。

5. IP数据包的分片与重组

IP数据包是在网络层进行数据传输的基本单位，但是它在传输中经常需要被分片与重组。熟悉IP数据包分片对于理解网络层与下面各层的关系，具有重要的意义。

本次练习的目的是：

(1) 掌握IP包分片的工作原理与涉及的相关字段。

(2) 理解网络层与数据链路层、物理层之间的关系。

6. IPv6数据包的封装与解析

IPv6协议是针对当前IP协议的问题制定的下一代协议标准。熟悉IPv6协议对于理解网络层协议的概念、层次结构与执行过程有很大的帮助。

本次练习的目的是:

(1) 掌握IPv6数据包头部中各个字段的含义与用途。

(2) 理解下一代网络层协议的设计思想与工作原理。

7. 发现网络中的活动主机

ICMP协议是TCP/IP协议族中的重要部分,用来补充IP协议缺少差错控制与查询机制的不足。熟悉ICMP数据包结构对于理解ICMP协议的工作原理有很大的帮助。

本次练习的目的是:

(1) 掌握ICMP数据包的各个字段的含义与用途。

(2) 理解ICMP协议的设计思想、工作原理与编程方法。

8. 发现服务器开启的TCP端口

端口是分布式进程通信在传输层使用的地址,网络服务需要在客户机与服务器的端口上提供。熟悉端口扫描技术对于理解传输层的工作原理有很大的帮助。

本次练习的目的是:

(1) 理解网络服务与端口的概念和相互关系。

(2) 掌握端口扫描技术的工作原理与编程方法。

9. TCP数据包的封装与发送

TCP协议是TCP/IP协议族的核心协议之一。熟悉TCP数据包结构对于理解网络层次结构,以及TCP协议与IP协议的关系,具有重要的意义。

本次练习的目的是:

(1) 掌握TCP数据包头部中各个字段的含义与用途。

(2) 理解传输层中TCP协议的设计思想与工作原理。

10. 基于TCP的客户机/服务器程序

网络服务采用客户机/服务器工作模式,TCP服务是需要建立连接的服务类型。熟悉基于TCP的客户机/服务器程序设计,对于理解有连接服务的设计思路有很大的帮助。

本次练习的目的是:

(1) 理解TCP服务的基本概念与主要功能。

(2) 掌握基于TCP的客户机/服务器程序设计方法。

11. 基于UDP的客户机/服务器程序

网络服务采用客户机/服务器工作模式,UDP服务是不需要建立连接的服务类型。熟悉基于UDP的客户机/服务器程序设计,对于理解无连接服务的设计思路有很大的帮助。

本次练习的目的是:

(1) 理解UDP服务的基本概念与主要功能。

(2) 掌握基于UDP的客户机/服务器程序设计方法。

12. FTP客户机程序设计

Internet中提供了很多类型的网络服务,这些服务实际上都是应用层的服务。熟悉文件传输服务的FTP客户机程序设计,对于理解应用层服务的编程方法有很大的帮助。

本次练习的目的是：

（1）掌握 FTP 服务的基本概念与工作原理。

（2）掌握应用层服务的基本设计思路与编程方法。

13. POP 客户机程序设计

Internet 中提供了很多类型的网络服务，这些服务实际上都是应用层的服务。熟悉电子邮件服务的 POP 客户机程序设计，对于理解应用层服务的编程方法有很大的帮助。

本次练习的目的是：

（1）掌握电子邮件服务的基本概念与工作原理。

（2）掌握应用层服务的基本设计思路与编程方法。

14. 包过滤防火墙程序设计

防火墙是网络安全技术中的重要组成部分。熟悉包过滤路由器的结构与工作原理，对于理解网络安全技术的设计思路有重要的意义。

本次练习的目的是：

（1）理解防火墙的基本概念与主要功能。

（2）掌握包过滤技术的设计思路与编程方法。

本书中的各个题目之间没有前后顺序的约束关系，读者可以根据自己的基础与兴趣独立地选择练习内容。表 1-1 总结了以上 14 个软件编程课题的题目、层次、练习目的与对应的难度级。

表 1-1 软件编程课题的题目、层次、练习目的与难度级

序号	编程课题的题目	层次	练习目的	难度级
1	Ethernet 帧的封装与解析	数据链路层	① 掌握 Ethernet 帧结构中各字段的含义与用途； ② 掌握 Ethernet 帧结构解析软件设计与编程方法	★
2	Ethernet 帧的 CRC 校验		① 掌握 Ethernet 帧校验的计算过程与用途； ② 理解数据链路层协议的设计思想与工作原理	★★
3	IP 地址的合法性判断	网络层	① 掌握 IPv4 地址的基本结构与分类方法； ② 理解网络层协议的设计思想与工作原理	★
4	IP 数据包的捕获与解析		① 掌握 IP 头部中各个字段的含义与用途； ② 掌握通过网卡截获 IP 包的基本方法	★★
5	IP 数据包的分片与重组		① 掌握 IP 包分片的工作原理与涉及的相关字段； ② 理解网络层与数据链路层、物理层之间的关系	★
6	IPv6 数据包的封装与解析		① 掌握 IPv6 包头部中各个字段的含义与用途； ② 理解下一代网络层协议的设计思想与工作原理	★★
7	发现网络中的活动主机		① 掌握重要的 ICMP 包结构中各个字段的含义与用途； ② 理解 ICMP 协议的设计思想与工作原理	★★★
8	发现服务器开启的 TCP 端口	传输层	① 理解网络服务与端口的概念与相互关系； ② 掌握端口扫描技术的工作原理与编程方法	★
9	TCP 数据包的封装与发送		① 掌握 TCP 包结构中各字段的含义与用途； ② 理解传输层中 TCP 协议的设计思想与工作原理	★
10	基于 TCP 的客户机/服务器程序		① 理解 TCP 服务的基本概念与主要功能； ② 掌握基于 TCP 的客户机/服务器程序设计方法	★★
11	基于 UDP 的客户机/服务器程序		① 理解 UDP 服务的基本概念与主要功能； ② 掌握基于 UDP 的客户机/服务器程序设计方法	★★

续表

序号	编程课题的题目	层次	练 习 目 的	难度级
12	FTP 客户机程序设计	应用层	① 掌握文件传输服务的基本概念与工作原理; ② 掌握应用层 FTP 协议的设计思想与编程方法	★★★
13	POP 客户机程序设计		① 掌握电子邮件服务的基本概念与工作原理; ② 掌握应用层 POP 协议的设计思想与编程方法	★★★
14	包过滤防火墙程序设计		① 理解防火墙的基本概念与主要功能; ② 掌握包过滤技术的设计思想与编程方法	★★

第 2 章 Socket 编程基础知识

2.1 Socket 编程的基本概念

网络环境中不同计算机之间的通信是分布式进程通信,它们采用的都是客户机/服务器工作模式。客户机与服务器都是进行通信的应用程序,它们分布在网络环境中的不同计算机上。

2.1.1 套接字的概念

在客户机/服务器工作模式中,客户机向服务器发出对服务的请求,服务器向客户机返回对请求的响应。为了完成不同计算机的应用进程之间的通信,TCP/IP 协议在全网范围内唯一地标识一个进程,这时需要使用网络层的 IP 地址与传输层的端口号,它们合起来就称为套接字(Socket)。图 2-1 给出了 Socket 地址的概念。如果客户机与服务器的进程之间进行通信,需要使用客户机的 Socket 与服务器的 Socket,这与 UNIX 网络编程中的五元组概念是一致的。

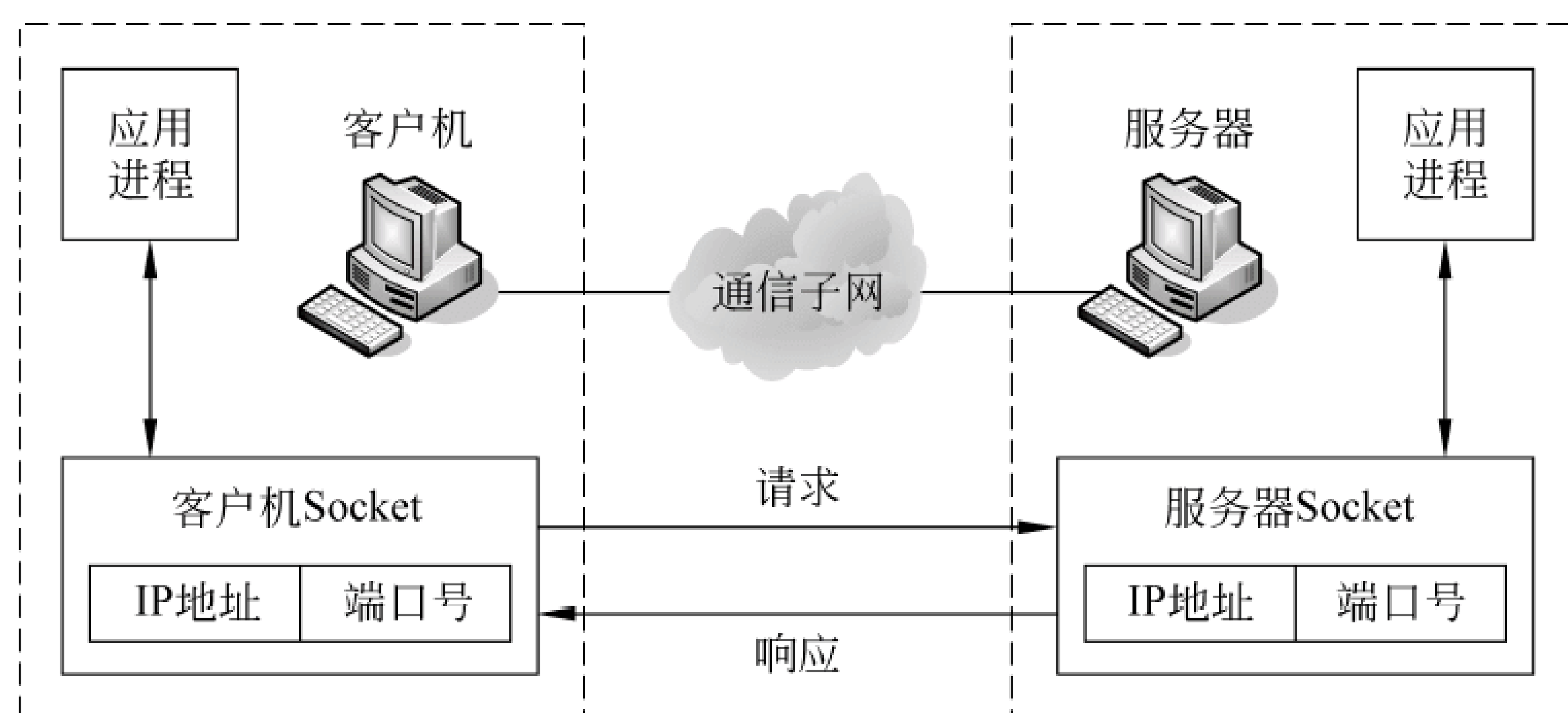


图 2-1 Socket 地址的概念

网络环境中的分布式进程通信需要采用 Socket 编程方式。网络环境中的每台计算机都有自己的操作系统,操作系统应该提供与网络应用程序的接口,即网络环境中的应用编程接口(Application Program Interface,API)。图 2-2 给出了网络编程接口的层次。实际上,Socket 最初是美国加州大学伯克利分校为 UNIX 操作系统开发的一种网络编程接口,称为 Berkeley Socket。Socket 屏蔽了底层通信软件和操作系统的差异,使任何两台安装 TCP/

IP协议软件和遵循套接字规范的计算机之间进行通信成为可能。随着UNIX操作系统与TCP/IP协议的广泛应用,Socket已经成为当前最流行的网络编程接口。

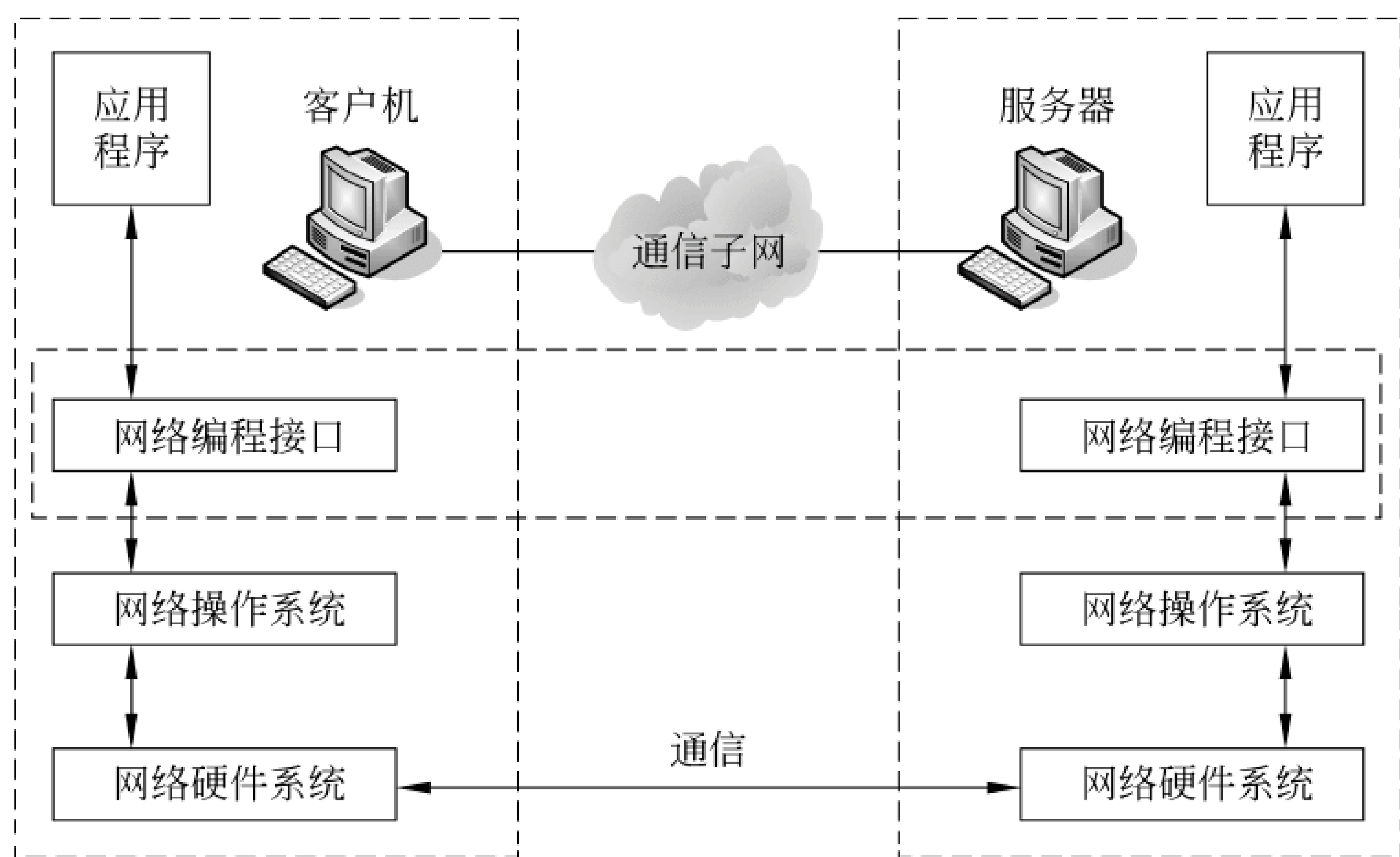


图 2-2　网络编程接口的层次

2.1.2　套接字的分类

Socket是一种独立于协议的网络编程接口,在OSI模型中主要集中在传输层与会话层。Socket定义了很多用于网络通信的函数与数据结构,程序员可以利用它们来开发TCP/IP网络中的应用程序。

Socket可以支持不同的通信协议与套接字类型。不同的通信协议针对的是不同类型的网络。Berkeley Socket支持多种类型的协议族,包括UNIX、TCP/IP、Xerox、Novell与AppleTalk等协议。最初版本的Winsock只支持IPv4协议,新版本的Winsock开始支持更多种协议。目前,比较常见的协议是IPv4(即常说的IP),它是一种广泛应用于Internet的网络层协议。

不同类型的Socket针对的是不同类型的网络应用。目前,常见的Socket类型主要有以下三种。

1. 流式套接字

流式套接字(Stream Socket)主要用于TCP协议。流式套接字提供了双向的、有序的、无重复的、无记录边界的数据流服务。图2-3给出了流式套接字的工作过程。流式套接字的设计是针对面向连接的服务,在数据传输之前需要预先建立数据传输连接,在数据传输结束后需要释放传输连接。同时,通信双方需要对传输数据进行验证,这样就可以保证数据传输的正确性。

2. 数据报套接字

数据报套接字(Datagram Socket)主要用于UDP协议。数据报套接字提供了双向的、无序的、可能重复的、有记录边界的数据流服务。图2-4给出了数据报套接字的工作过程。数据报套接字的设计针对的是无连接的服务,在数据传输之前不需要建立数据传输连接,它无法保证数据传输的正确性。

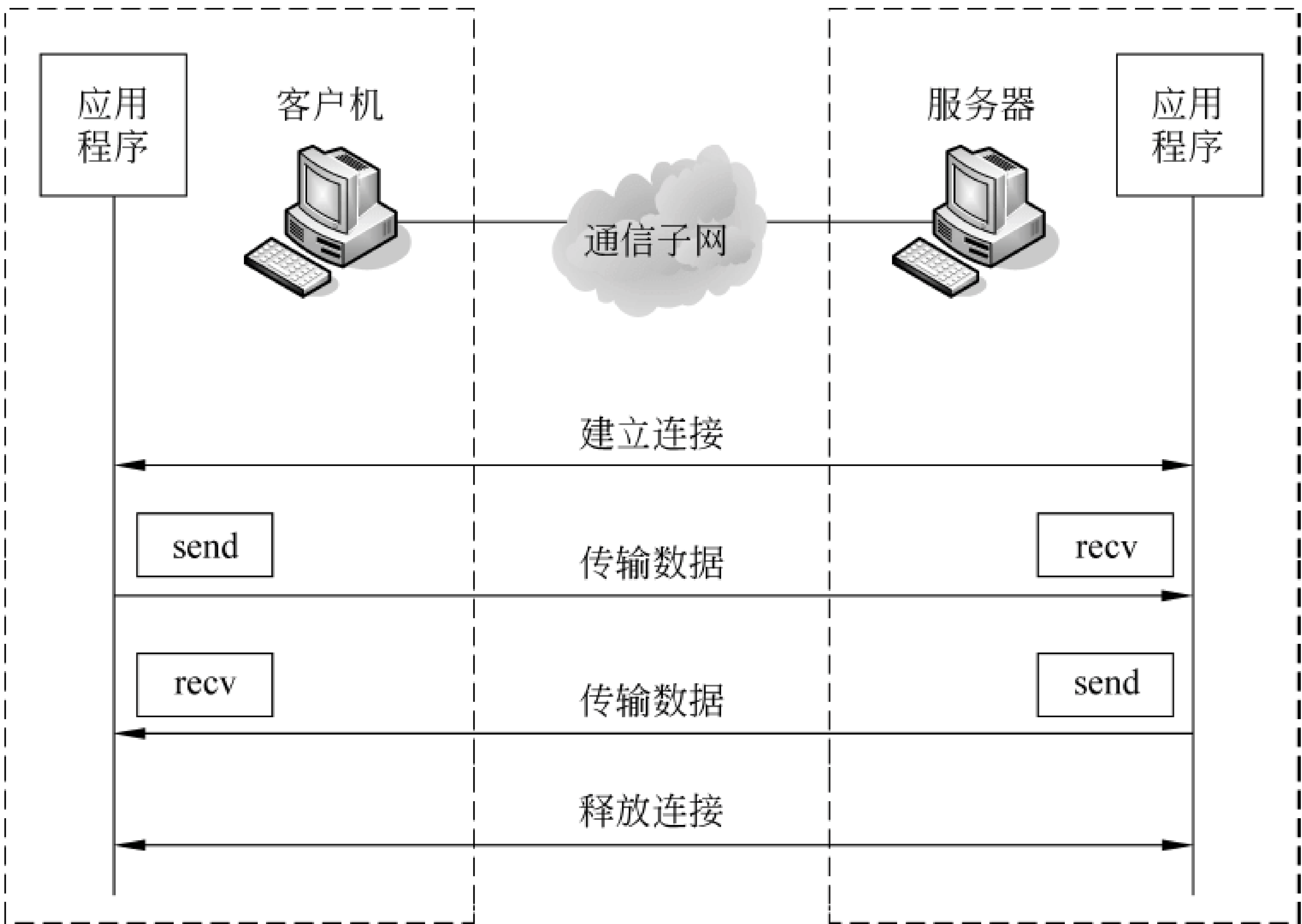


图 2-3　流式套接字的工作过程

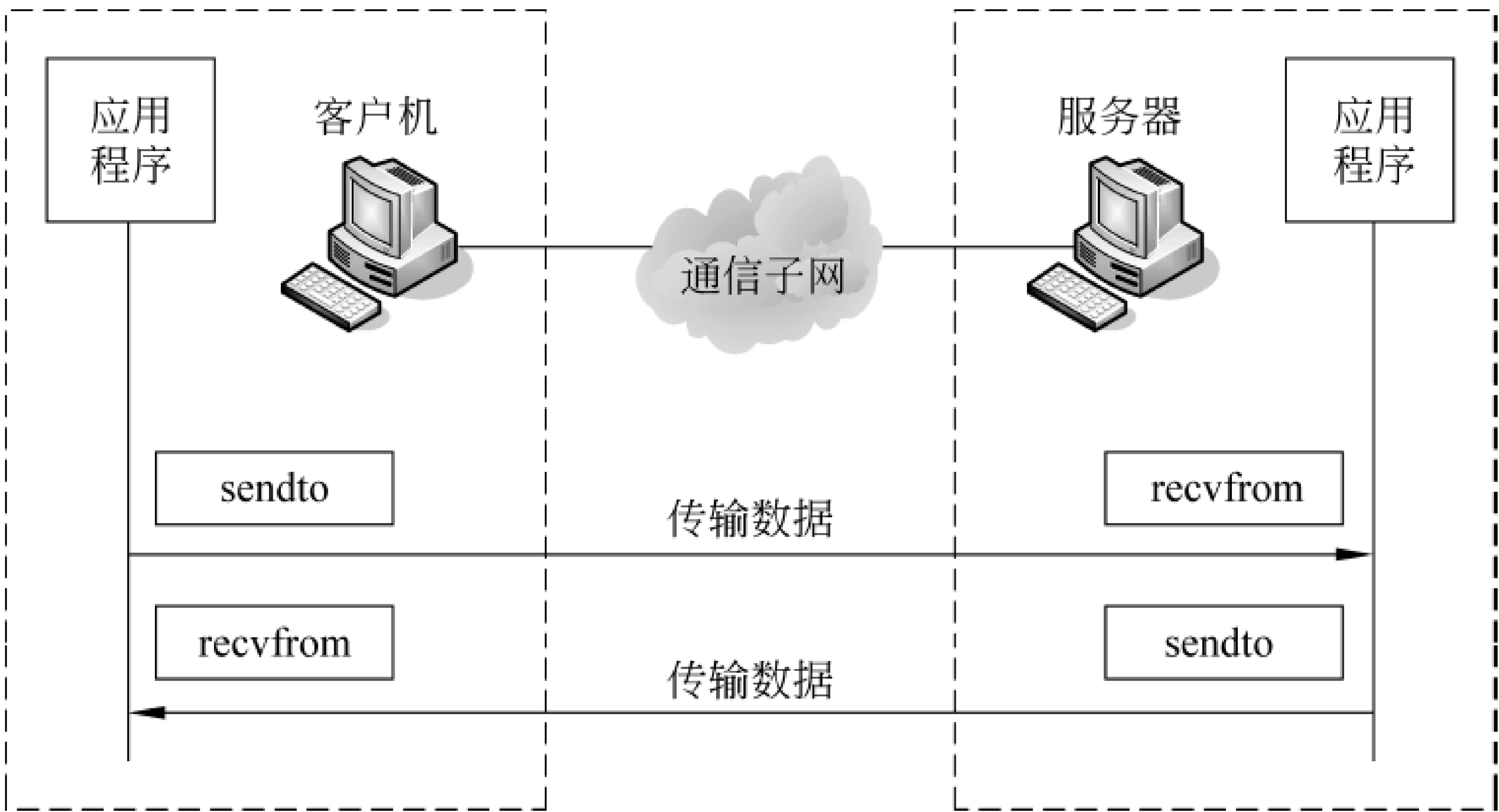


图 2-4　数据报套接字的工作过程

3. 原始套接字

原始套接字(Raw Socket)主要用于访问底层协议,例如 IP、ICMP 与 IGMP 等协议。原始套接字可以保存 IP 数据包中的完整 IP 头部;前面两种套接字不保留 IP 数据包中的 IP 头部,而只是将接收的 IP 数据包存储、转发或丢弃。如果我们要对 IP 数据包的头部进行分析,只能使用原始套接字来进行网络编程。

2.2　Winsock 网络编程接口

20 世纪 90 年代初期,Microsoft 公司联合其他几家公司制定了 Windows 下的网络编程接口,即 Windows Sockets 规范(简称 Winsock),使 Windows 环境中的 Socket 应用开发规范化与标准化。

2.2.1 Winsock的基本概念

1. Winsock的工作方式

Winsock规范并不是一种实际的网络协议,它是TCP/IP协议在Windows系统中的封装。Winsock不仅提供标准Berkeley Socket功能的调用集,而且针对Windows的特点进行必要的扩充,可以与不同操作系统的计算机进行Socket通信。通过调用Winsock的接口函数可以实现基本的Socket功能。那些扩充的功能调用以WSA(Windows Sockets Asynchronous)为前缀,表明它们都支持异步模式的I/O操作,并采用符合Windows消息机制的网络事件异步选择机制。

Windows操作系统中的Socket是以DLL的形式实现的。从Windows 95开始,操作系统内置的是Winsock 1.1。后来过渡到Windows XP,其内置的Winsock开始更新为Winsock 2.2。其中,Winsock 1.1的DLL为Winsock.dll,而Winsock 2.2的DLL为Wsock32.dll。在支持Winsock 2.2的系统中,Winsock 1.1函数调用自动地由Wsock32.dll映射到Winsock.dll。Winsock提供两种I/O方式:同步方式与异步方式。其中,同步方式又称为阻塞方式,异步方式又称为非阻塞方式。

在阻塞方式中,这类Socket函数被调用后,在完成其任务之前不会返回。在该函数调用返回之前,该Socket不能进行其他操作,调用它的进程处于挂起状态,因此这种工作模式被称为阻塞方式。例如,应用程序调用send()或recv()函数后,需要花费相当长的时间等待数据到达,该进程在这段时间内无法继续执行。如果发送方的数据无法到达,该进程就会无限期地等待下去。Berkeley Socket的很多函数都是阻塞方式。图2-5给出了阻塞方式的工作原理。

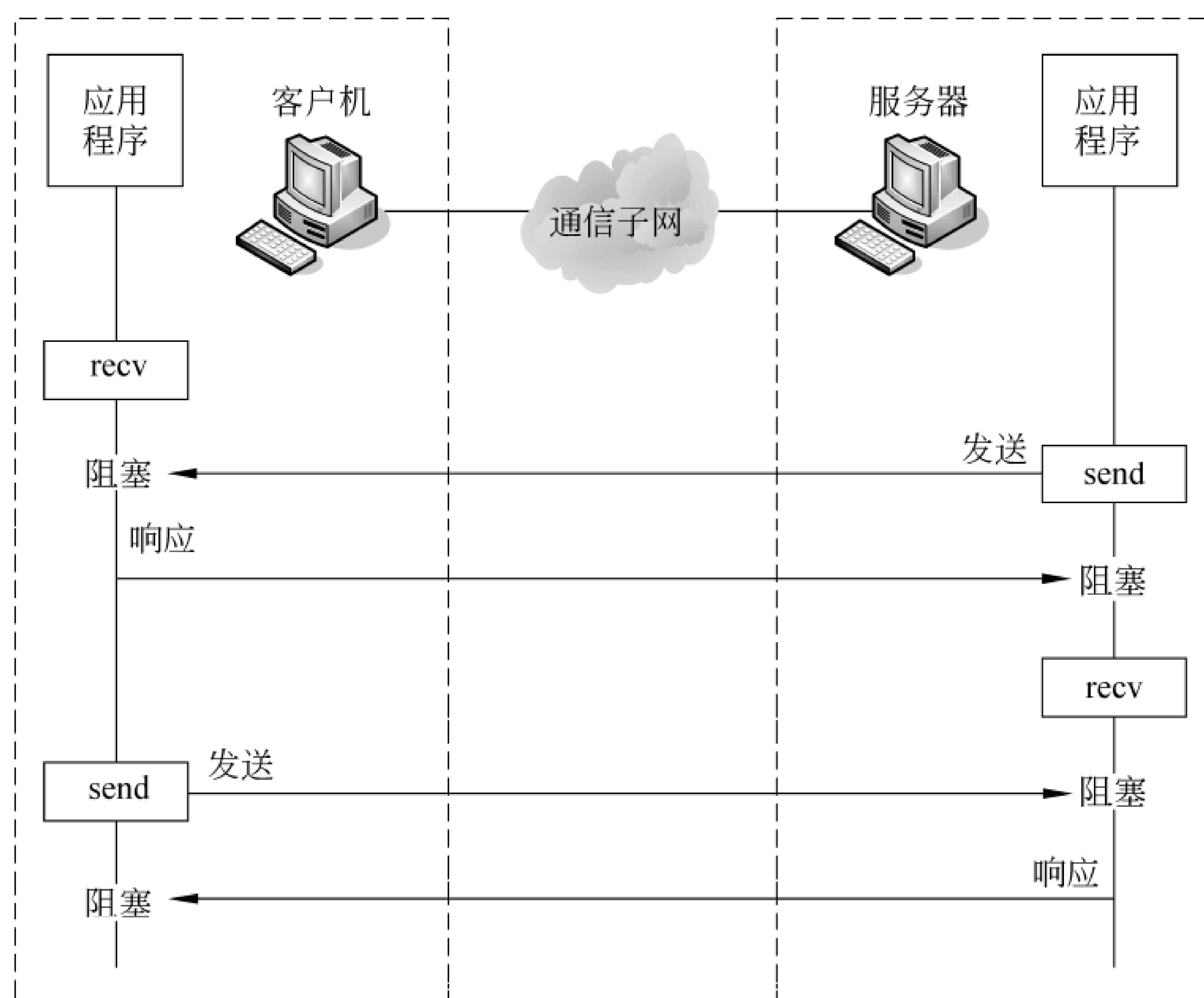


图2-5 阻塞方式的工作原理

2. Winsock 异步 I/O 模型

Winsock 在保留基本 Socket 函数的基础上，通过异步机制管理一个或多个 Socket 上的通信。在非阻塞方式中，这类 Socket 函数被调用后，该函数不等任务完成就会立即返回，调用该函数的进程可以继续执行。当 Socket 函数所要执行的任务完成后，Winsock 的 DLL 向应用程序发送一个事先约定好的消息，应用程序根据这个消息做出相应处理。表 2-1 给出了 Winsock 提供的异步 I/O 模型。Winsock 1.1 提供了选择与异步选择模型，Winsock 2.2 提供了表中列出的 5 种异步 I/O 模型。

表 2-1　Winsock 提供的异步 I/O 模型

I/O 模型	说　　明
选择模型	Select()函数为核心的阻塞模式
异步选择模型	WSAAsynSelect()函数为核心，在窗口上实现数据读写的异步通知
事件选择模型	WSAEventSelect()函数为核心，在事件上实现数据读写的异步通知
重叠 I/O 模型	创建一个重叠数据结构，一次投递一个或多个 I/O 请求
完成端口模型	创建一个完成端口对象，通过指定数量的线程来管理重叠 I/O

异步选择模型是一种常用的异步 I/O 模型。应用程序可以在一个套接字上接收以 Windows 消息为基础的事件通知。该模型的实现方法是通过调用 WSAAsynSelect()函数，自动地将套接字设置为非阻塞模式，向 Windows 系统注册一个或多个网络事件，并提供一个通知时使用的窗口句柄。当注册的网络事件发生时，对应的窗口将收到一个基于消息的通知。用户可以通过该函数注册感兴趣的网络事件，例如接收缓冲区满、允许发送数据、请求建立连接等。WSAAsynSelect()函数的原型为：

```
int WSAAsynSelect(SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent)
```

其中，s 指定应用程序使用的 Socket；hWnd 指定接收 Winsock 消息的窗口句柄；wMsg 指定向窗口 hWnd 提交的消息名称；lEvent 指定经注册的网络事件，它可以是一种事件，也可以是几种事件的组合。表 2-2 给出了 WSAAsynSelect 可选择的网络事件。该函数执行成功后返回 0，失败后则返回 SOCKET_ERROR。

表 2-2　WSAAsynSelect 可选择的网络事件

事件符号	说　　明
FD_READ	Socket 接收数据的消息
FD_WRITE	Socket 发送数据的消息
FD_ACCEPT	Socket 接收到连接请求的消息
FD_CONNECT	Socket 连接成功的消息
FD_CLOSE	Socket 连接关闭的消息
FD_OOB	Socket 接收到紧急数据的消息

2.2.2　初始化与卸载 Winsock

1. WSAStartup()函数

WSAStartup()函数的作用是初始化 Winsock 的 DLL。当应用程序调用 WSAStartup()函

数时,操作系统根据请求的 Socket 版本搜索相应的 Socket 库,并将应用程序与找到的 Socket 库进行绑定,然后应用程序就可以调用 Socket 库中的其他函数。在 Winsock 的 DLL 内部维持着一个计数器,仅在第一次调用 WSAStartup()时真正装载 DLL,以后每调用一次只是将计数器加 1。

WSAStartup()函数的原型为:

```
int WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData)
```

其中,wVersionRequested 指定应用程序请求使用的 Socket 版本,高位字节指明副版本、低位字节指明主版本,目前 Winsock 主要有 1.1 和 2.2 版本,因此该参数可以是 0x101 或 0x202;lpWSAData 用来返回请求的 Socket 版本信息。该函数执行成功后返回 0,失败后则返回 SOCKET_ERROR。

2. WSACleanup()函数

WSACleanup()函数的作用是卸载 Winsock 的 DLL。当应用程序调用 WSACleanup()函数时,操作系统会解除应用程序与 Socket 库的绑定,并且释放 Socket 库占用的系统资源。

WSACleanup()函数的原型为:

```
int WSACleanup()
```

WSACleanup()函数的功能与 WSAStartup()相反,每调用一次只是将计数器减 1,当计数器减到 0 时将 DLL 从内存中卸载。因此,调用 WSACleanup()与调用 WSAStartup()的次数应该相同。该函数执行成功后返回 0,失败后则返回 SOCKET_ERROR。

另外,在使用 Winsock 2.0 进行网络编程时,需要在程序开始部分包含 winsock2.h 头文件,并加载 ws2_32.lib 库文件与 ws2_32.dll 动态链接库。

2.2.3 基本 Socket 函数

Winsock 是以 Berkeley Socket 为基础定义的,因此它们大多数函数的使用方法基本相同,并且在函数设计上也比较类似。

1. 创建与关闭套接字

(1) socket()函数

socket()函数的作用是创建套接字。当应用程序调用该函数时,操作系统会为应用程序创建指定类型的套接字,并为该套接字分配合适的系统资源。在 Windows 扩展模式下,对应的函数是 WSASocket()。

socket()函数的原型为:

```
SOCKET socket(int af,int type,int protocol)
```

其中,af 指定通信协议类型,通常设为 AF_INET;type 指定创建的套接字类型,流式套接字为 SOCK_STREAM,数据报套接字为 SOCK_DGRAM,原始套接字为 SOCK_RAW;protocol 依赖于第二个参数,指定套接字使用的协议,通常设为 IPPROTO_IP。该函数执行

成功后返回新创建的套接字描述符，失败后则返回 SOCKET_ERROR。

套接字描述符是一个整数类型的值。每个进程空间中都有一个套接字描述符表，该表存放套接字描述符和套接字数据结构的对应关系。

(2) closesocket()函数

closesocket()函数的作用是关闭套接字。当应用程序调用 closesocket()时，操作系统会关闭套接字并释放相应的系统资源。

closesocket()函数的原型为：

```
int closesocket(SOCKET s)
```

其中，s 指定要关闭的套接字描述符。该函数执行成功后返回 0，失败后则返回 SOCKET_ERROR。

2. 绑定套接字

bind()函数的作用是绑定本地地址与套接字。当应用程序按要求创建一个套接字后，套接字结构中会有一个默认的 IP 地址和端口号。无论是服务器端还是客户端程序，都需要将本地地址绑定到新创建的套接字上。

bind()函数的原型为：

```
int bind(SOCKET s,const struct sockaddr * name,int namelen)
```

其中，s 指定要绑定的套接字描述符；name 指定 sockaddr 结构的套接字地址，通常使用 sockaddr_in 结构，使用时可将它强制转换为 sockaddr 结构；namelen 指定套接字地址结构的长度。该函数执行成功后返回 0，失败后则返回 SOCKET_ERROR。

3. 与服务器建立连接

(1) listen()函数

listen()函数的作用是监听端口的连接建立请求。listen()函数是专门为流式套接字设计的，用于有连接的 TCP 服务类型。服务器端程序调用 listen()函数使流式套接字处于监听状态。

listen()函数的原型为：

```
int listen(SOCKET s,int backlog)
```

其中，s 指定要监听的套接字描述符；backlog 指定流套接字要维护的客户连接请求队列，该队列最多能容纳 backlog 个客户连接请求。该函数执行成功后返回 0，失败后则返回 SOCKET_ERROR。

(2) connect()函数

connect()函数的作用是请求与服务器建立连接。connect()函数是专门为流式套接字设计的，用于有连接的 TCP 服务类型。客户端程序调用 connect()函数向服务器端 Socket 发出建立连接请求。在 Windows 扩展模式下，对应的函数是 WSAConnect()。

connect()函数的原型为：

```
int connect(SOCKET s,const struct sockaddr * name,int namelen)
```

其中,s 指定客户端的套接字描述符;name 返回服务器端的套接字地址结构;namelen 返回服务器端的套接字地址长度。该函数执行成功后返回 0,失败后则返回 SOCKET_ERROR。

(3) accept()函数

accept()函数的作用是对连接建立请求的响应。accept()函数是专门为流式套接字设计的,用于有连接的 TCP 服务类型。服务器端程序调用 accept()函数从处于监听状态的流式套接字的客户连接请求队列中取出排在最前面的一个客户请求,并且创建一个新的套接字来与客户端套接字建立连接。在 Windows 扩展模式下,对应的函数是 WSAAccept()。

accept()函数的原型为:

```
SOCKET accept(SOCKET s,struct sockaddr * addr,int addrlen)
```

其中,s 指定要监听的套接字描述符;addr 返回新创建的套接字地址结构;addrlen 返回新创建的套接字地址长度。该函数执行成功后返回新创建的套接字描述符;失败后则返回 SOCKET_ERROR。此后,与客户端通信需要使用新创建的套接字。

4. 发送与接收数据

(1) send()与 sendto()函数

send()与 sendto()函数的作用都是发送数据。无论是服务器端还是客户端程序,都需要使用 send()或 sendto()函数向对方发送数据。其中,send()函数是专门为流式套接字设计的,用于有连接的 TCP 服务类型;sendto()函数是为数据报套接字设计的,用于无连接的 UDP 服务类型。在 Windows 扩展模式下,对应的函数分别是 WSASend()与 WSASendTo()。

这两个函数的原型分别为:

```
int send(SOCKET s,const char * buf,int len,int flags)
int sendto(SOCKET s,const char * buf,int len,int flags,const char * to,int tolen)
```

在 send()函数中,s 指定发送端套接字描述符;buf 指定发送端等待发送数据的缓冲区;len 指定要发送数据的字节数;flags 指定需要附加的标志位,通常设置为 0。在 sendto()函数中,前 4 个参数与 send()函数相同;to 指定存放接收端等待接收数据的缓冲区;tolen 指定要接收数据的字节数。这两个函数执行成功后返回发送数据的字节数,失败后则返回 SOCKET_ERROR。

(2) recv()与 recvfrom()函数

recv()与 recvfrom()函数的作用都是接收数据。无论是服务器端还是客户端程序,都需要使用 recv()或 recvfrom()函数从对方接收数据。其中,recv()函数是专门为流式套接字设计的,用于有连接的 TCP 服务类型;recvfrom()函数是为数据报套接字设计的,用于无连接的 UDP 服务类型。在 Windows 扩展模式下,对应的函数分别是 WSARecv()与 WSARecvFrom()。

这两个函数的原型分别为:

```
int recv(SOCKET s,char * buf,int len,int flags)
int recvfrom(SOCKET s,char * buf,int len,int flags,const char * from,int fromlen)
```

在 recv()函数中，s 指定接收端套接字描述符；buf 指定接收端等待接收数据的缓冲区；len 指定要接收数据的字节数；flags 指定需要附加的标志位，通常设置为 0。在 recvfrom()函数中，前 4 个参数与 recv()函数相同；from 指定存放发送端等待发送数据的缓冲区；fromlen 指定要发送数据的字节数。这两个函数执行成功后返回接收数据的字节数，失败后则返回 SOCKET_ERROR。

5. 读取与设置 Socket 属性

getsockopt()函数的作用是读取套接字的属性。setsockopt()函数的作用是设置套接字的属性。

这两个函数的原型分别为：

```
int getsockopt(SOCKET s,int level,int optname,char * optval,int * optlen)
int setsockopt(SOCKET s,int level,int optname,const char * optval,int optlen)
```

其中，s 指定要读取或设置属性的套接字；level 指定套接字选项的级别，大多数是特定协议和套接字专有，例如 IP 协议应为 IPPROTO_IP；optname 指定要读取或设置属性的选项名称，例如 SO_RCVTIMEO 表示接收超时，SO_SNDTIMEO 表示发送超时等；optval 指定存放选项值的缓冲区指针；optlen 指定该缓冲区的长度。这两个函数执行成功后返回 0，失败后则返回 SOCKET_ERROR。

6. 字节序转换函数

在 Internet 中存在多种类型的网络与计算机，不同类型的计算机表示数据的字节顺序不同。16 位整数在内存中有两种存储方式：高位字节在前与低位字节在前。这两种存储方式分别称为 big-endian 与 little-endian。例如，“B135”可存储为“B1 35”或“35 B1”。如果在网络之间传输的数据没有统一，网络传输的另一方虽然获得正确的数据，但是可能因为理解的差异而造成数据错误。

网络协议中的数据采用统一的网络字节序，Internet 规定的网络字节序是 big-endian。例如，sockaddr_in 必须以网络字节序表示，而不能直接使用本机字节序的值。从程序可移植的角度，即使本机的内部字节序与网络字节序相同，也应该在传输数据之前调用字节序转换函数，以保证程序移植到其他计算机能正确执行。

Socket 提供了以下字节序转换函数。

(1) unsigned long htonl(unsigned long hostlong)：将无符号长整型数从主机字节顺序转换为网络字节序。其中，hostlong 是无符号长整型数。

(2) unsigned long ntohl(unsigned long netlong)：将无符号长整型数从网络字节序转换为主机字节顺序。其中，netlong 是无符号长整型数。

(3) unsigned short htons(unsigned short hostshort)：将无符号短整型数从主机字节顺序转换为网络字节序。其中，hostshort 是无符号短整型数。

(4) unsigned short ntohs(unsigned short netshort)：将无符号短整型数从网络字节序转换为主机字节顺序。其中，netshort 是无符号短整型数。

7. 其他相关函数

Winsock 提供了很多辅助性的函数。

(1) in_addr long inet_addr(const char * cp)：将点分十进制 IP 地址转换为 in_addr 结

构 IP 地址(unsigned long 类型)。其中,cp 是点分十进制 IP 地址。

(2) char * inet_ntoa(struct in_addr in):将 in_addr 结构 IP 地址转换为点分十进制 IP 地址。其中,in 是 in_addr 结构 IP 地址。

(3) int gethostname(char * name,int namelen):获取主机名。其中,name 是存放主机名的缓冲区,namelen 是缓冲区大小。

(4) struct hostent * gethostbyname(const char * name):根据主机名获取主机信息(主机名、别名与地址等)。其中,name 是主机名。

(5) struct hostent * gethostbyaddr(const char * addr, int len, int type):根据 IP 地址获取主机信息(主机名、别名与地址等)。其中,addr 是点分十进制的 IP 地址,len 是地址长度,type 是地址类型。

(6) struct protoent * getprotobyname(const char * name):根据协议名称获取协议信息(协议名、别名与端口号)。其中,name 是协议名。

(7) struct protoent * getprotobynumber(int proto):根据端口号获取协议信息(协议名、别名与端口号)。其中,proto 是协议端口号。

8. GetLastError()函数

GetLastError()函数用来获得错误类型。如果某个 Socket 函数返回 SOCKET_ERROR,说明该函数在处理过程中出现错误,这个错误可能由不同原因而引起。调用 GetLastError()可获得该错误对应的类型码,判断错误类型以便决定如何处理问题。表 2-3 给出了主要的错误类型码。在异步模式下,对应的函数是 WSAGetLastError()。

表 2-3 主要的错误类型码

错误类型码	错 误 类 型
10013	执行访问权限不支持的套接字操作
10014	通过指针参数指向非法的指针地址
10022	在套接字操作中提供无效的参数
10035	无法立即完成非阻塞套接字操作
10036	在套接字上已有阻塞操作正在执行
10040	数据报套接字的消息超过缓冲区大小
10045	对象类型不支持尝试的套接字操作
10051	向无法连接的网络尝试套接字操作
10052	在操作中由于故障导致连接失败
10053	主机中的软件自动放弃已建立的连接
10054	远程主机强迫关闭已建立的连接
10055	缓冲区不足导致无法进行套接字操作
10060	远程主机没有响应导致连接尝试失败
10061	远程主机主动拒绝导致连接无法建立
10065	向无法连接的主机尝试套接字操作
10092	不支持请求的 Windows 套接字版本

2.2.4 套接字地址结构

1. sockaddr 结构

sockaddr 结构是一种通用的 Socket 地址结构，可用于不同类型的协议族，例如 TCP/IP、UNIX、AppleTalk 等。sockaddr 结构随着选择的协议而变化。

下面给出的是 sockaddr 结构：

```
struct sockaddr
{
    short sa_family;
    char sa_data[14];
};
```

其中，sa_family 指定主机地址类型（TCP/IP 协议族是 AF_INET，UNIX 协议族是 AF_UNIX，Xerox 协议族是 AF_NS），这里通常使用 AF_INET；sa_data 指定 14 字节的协议地址，包含该 Socket 结构的 IP 地址与端口号。

2. sockaddr_in 结构

sockaddr_in 结构是更常用的 Socket 地址结构，专门应用于 TCP/IP 类型的协议族。在进行实际的网络编程时，通常将 sockaddr_in 转化为 sockaddr 结构。

下面给出的是 sockaddr_in 结构：

```
struct sockaddr_in
{
    short sin_family;
    short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

其中，sin_family 指定主机使用的地址类型，TCP/IP 协议族为 AF_INET；sin_port 指定协议使用的端口号；sin_addr 指定主机使用的 IP 地址，即 in_addr 结构的 IP 地址，如果将 sin_addr 设置为 INADDR_ANY，则表示任意 IP 地址；sin_zero 是填充字段，将 sockaddr_in 转化为 sockaddr 结构时，通过填充 0 保证与 sockaddr 大小一致。

3. hostent 结构

hostent 结构是表示主机信息的数据结构。由于主机的 IP 地址难以记忆与读写，因此通常使用主机名来表示主机，这样需要在 IP 地址与主机名之间对应。例如，函数 gethostbyname()就是一种对应函数。

下面给出的是 hostent 结构：

```
struct hostent
{
    char * h_name;
    char **h_aliases;
```

```
    short h_addrtype;
    short h_length;
    char **h_addr_list;
};
```

其中,h_name 指定主机使用的主机名;h_aliases 指定主机使用的别名指针;h_addrtype 指定主机使用的地址类型;h_length 指定主机使用的地址长度;h_addr_list 指定主机使用的地址指针。

4. protoent 结构

protoent 结构是表示协议信息的数据结构。由于协议的名称是给人来理解的,而计算机通常使用的是端口号,因此需要在协议名与端口号之间进行对应。例如,函数 getprotobyname()就是一种对应函数。

下面给出的是 protoent 结构:

```
struct protoent
{
    char * p_name;
    char **p_aliases;
    short p_proto;
};
```

其中,p_name 指定主机使用的协议;p_aliases 指定协议使用的别名指针;p_proto 指定协议使用的端口号。

5. TCP/IP 协议定义

在针对 TCP/IP 协议族的网络编程中,可能对某种具体的网络协议进行处理。根据 TCP/IP 协议的规定,每种网络层协议都有其对应的值。例如,IP 协议对应的值为 0,而 TCP 协议对应的值为 6。Socket 接口针对每种协议也进行了定义。

下面给出的是常见的 TCP/IP 协议的定义:

```
#define IPPROTO_IP 0
#define IPPROTO_ICMP 1
#define IPPROTO_IGMP 2
#define IPPROTO_TCP 6
#define IPPROTO_UDP 17
#define IPPROTO_RAW 255
```

第 3 章 Ethernet 帧的封装与解析

3.1 设计目的

帧是在数据链路层进行数据传输的基本单位。熟悉帧结构，对于理解网络协议的概念、网络层次结构、协议执行过程以及网络问题处理方法，具有重要的意义。本章练习的目的是根据数据链路层的基本原理，通过封装标准格式的 Ethernet 帧，了解 Ethernet 帧结构中各字段的含义与用途，从而深入理解网络协议的工作原理。

3.2 相关知识

本章涉及的相关知识包括数据链路层的概念与 Ethernet 帧结构。

3.2.1 数据链路层的概念

网络中的两台主机之间通信要遵循相同的网络协议。各种异构网络的互联带来了协议的标准化问题，这就促进了网络体系结构与参考模型的研究。OSI 参考模型是由 ISO 组织制定的一个网络互联参考模型。OSI 参考模型采用的是一种分层的体系结构，要求各个节点具有相同的层次，不同节点的相同层次具有相同功能，每层使用下层提供的服务并向上层提供服务。图 3-1 给出了 OSI 参考模型的结构。OSI 参考模型从下至上依次是：物理层、数据链路层、网络层、传输层、会话层、表示层与应用层。其中，应用层是 OSI 参考模型的最高层，主要描述各种高层网络应用及其协议。

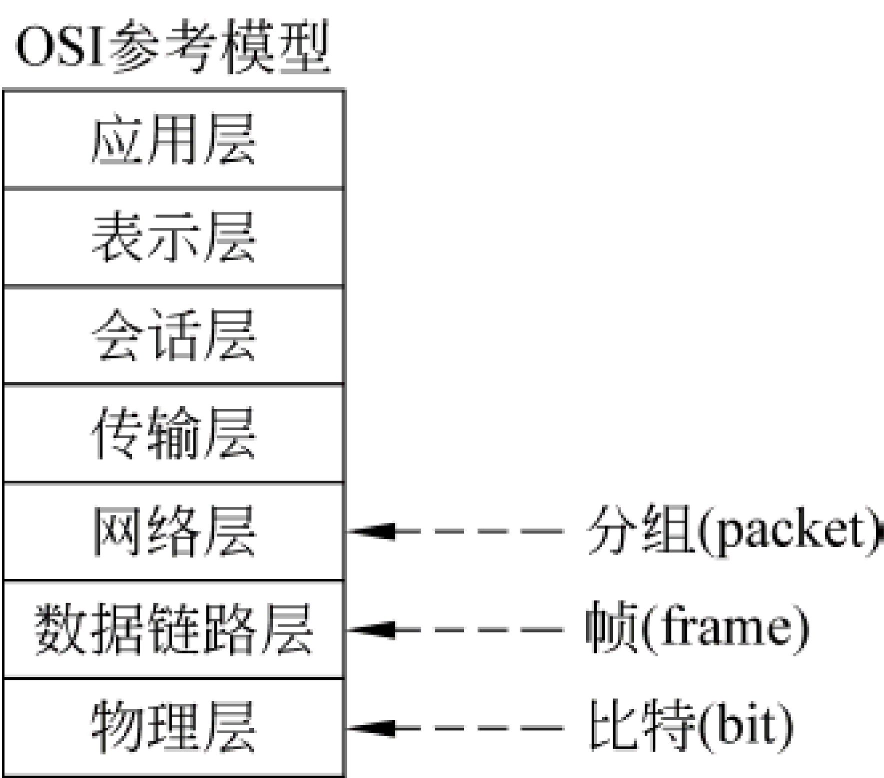


图 3-1　OSI 参考模型的结构

根据 OSI 参考模型的规定，数据链路层（data link layer）是参考模型的第 2 层。数据链路层的主要功能是：在底层的物理层提供的服务基础上，在作为通信实体的主机之间建立数据链路，在这个链路上传输以帧为单位的数据包，并采取差错控制与流量控制的方法，将有差错的物理连接变成无差错的数据链路。因此，数据链路层需要为上面的网络层提供服务，屏蔽各种物理网络以及传输介质的差异性。OSI 参考模型的每层都有各自的数据传输单位，经过不同层时都需要组装成相应的传输单位。帧（frame）是在数据链路层进

行数据传输的基本单位。

数据链路层包括很多种实际的物理网络,涵盖广域网、城域网、局域网等网络类型,它们用于不同覆盖范围的计算机组网。其中,局域网的覆盖范围是周边的有限范围。例如,一个校园、一栋大楼或一个实验室,将其中的计算机、终端与外设互连起来。1980年,IEEE成立致力于局域网标准化的IEEE 802委员会,制定了针对不同类型局域网的IEEE 802标准。其中,IEEE 802.3标准是以太网(Ethernet)的协议标准,该标准包括数据链路层与物理层两部分。数据链路层标准主要描述介质访问控制问题;物理层标准主要描述使用的传输介质,例如双绞线、光纤、同轴电缆等。

3.2.2 Ethernet帧的结构

术语"帧"最早来源于串行线路上的通信。发送节点在发送数据的前后各添加特殊的字符构成帧,这些特殊的字符称为帧头与帧尾。在某种程度上,Ethernet可以视为网络节点之间的数据链路层连接。IEEE 802.3标准在Ethernet V2.0规范的基础上制定,但是Ethernet V2.0和IEEE 802.3的Ethernet帧在结构上有一些差别。本书将按IEEE 802.3标准的帧结构进行讨论。图3-2给出了Ethernet帧结构。帧的基本长度单位是字节(byte,B)。

前导码 (7B)	帧前定界符 (1B)	目的地址 (2B/6B)	源地址 (2B/6B)	类型 (2B)	数据 (46B~1500B)	帧校验字段 (4B)

图3-2 Ethernet帧结构

IEEE 802.3标准的Ethernet帧结构由下面6部分组成。

1. 前导码与帧前定界符

前导码由56位(bit,b)即7B的10101010…10101010比特序列组成,换算成十六进制就是7个连续的0xaa。从物理层的角度来看,从网卡开始接收数据到进入稳定状态需要一定的时间。前导码的设计目的是保证网卡在帧的目的地址到来之前达到稳定状态。帧前定界符可以视为前导码的延续。帧前定界符由8位(即1B)的10101011比特序列组成,换算成十六进制就是1个0xab。

如果将前导码与帧前定界符放一起看,则是在62位的10101010…10比特序列后出现11,在这个11后面出现的是目的地址字段。也就是说,在帧开始时出现前导码与帧前定界符,则说明这是一个合法的帧。前导码与帧前定界符主要起接收同步作用,这8B的数据在接收后不需要保留,也不计入帧头长度字段值中。

2. 目的地址与源地址

目的地址与源地址分别表示帧的接收节点与发送节点的硬件地址。在网络节点之间进行通信需要使用硬件地址,这里使用的是网卡的MAC地址。这个地址在出厂时就固化在网卡的EPROM中,并保证该地址在全球范围内是唯一的。无论网卡被连接在哪个局域网中,也无论计算机被移动到哪个位置,网卡的MAC地址都是固定不变的。MAC地址是一个数据链路层地址,通常将它称为"物理地址"。IP地址是一个网络层地址,可由网管人员分配和通过软件设置,通常称为"逻辑地址"。

在研究Ethernet帧结构时,主要讨论以下两个问题。

(1) 目的地址和源地址长度可以是2B或6B。早期的Ethernet曾使用过2B的地址。

目前，Ethernet 都使用 6B(即 48b)的地址。MAC 地址由两个部分组成：公司唯一标识(OUI)与扩展唯一标识(EUI)。其中，OUI 的长度是 3B，它是分配给网卡制造商的编码；EUI 的长度是 3B，它可由网卡制造商来分配。图 3-3 给出了 MAC 地址的表示方法。MAC 地址由十六进制数用连字符分隔开表示。例如，08-01-00-2A-10-C3。

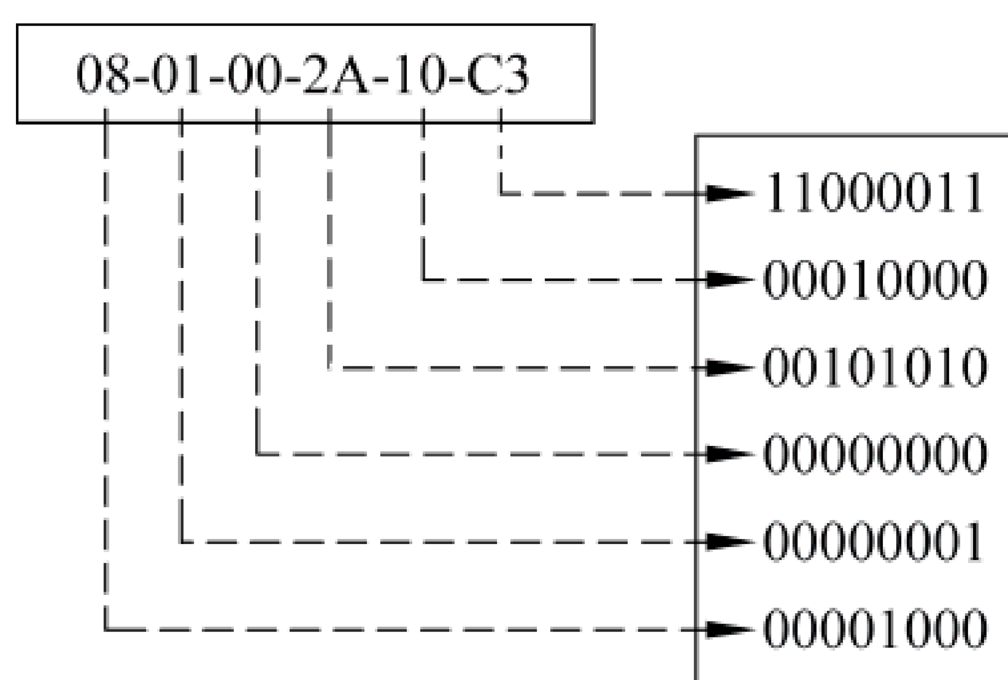


图 3-3　MAC 地址的表示方法

(2) Ethernet 帧的目的地址有三种类型：单播地址(unicast address)、多播地址(multicast address)与广播地址(broadcast address)。其中，目的地址的第 1 位为 0 则表示单播地址，第 1 位为 1 则表示多播地址，目的地址为全 1 则表示广播地址。如果目的地址是单播地址，表示该帧仅被自己的 MAC 地址与目的地址相同的网络节点接收。

3. 长度字段

长度字段(2B)表示数据字段包含的字节数。IEEE 802.3 标准规定 Ethernet 帧的数据部分最小长度为 46B，最大长度为 1500B。由于帧头长度为 18B(前导码与帧前定界符不计入长度)，因此 Ethernet 帧的最小长度为 64B，最大长度为 1518B。最小长度的设计目的是使接收节点有足够的时间检测到冲突，并及时通知源节点重新传输没有成功接收的帧。

4. 数据字段

数据字段用于保存发送给目的节点的实际数据。由于数据字段的最小长度为 46B，如果一个帧的数据部分少于 46B，则应将数据字段填充至 46B。填充字符可以是任意的字符，在实际应用中经常用 0 完成填充。但是，填充部分不计入长度字段值中。另外，数据字段的最大长度为 1500B。

5. 帧校验字段

帧校验字段(4B)用于判断帧在传输过程中是否出错。IEEE 802.3 标准规定的校验范围包括目的地址、源地址、长度与数据字段。前导码与帧前定界符不需要进行校验。帧校验采用 32 位的 CRC 校验(CRC-32)。CRC-32 的生成多项式为：

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

另外，在某些帧结构中还会包括帧类型字段，用来识别该帧所承载的数据类型。当一个帧到达目的节点时，根据帧类型决定用哪个协议软件模块进行处理。

3.3 例题分析

3.3.1 设计要求

根据给出的 IEEE 802.3 格式的 Ethernet 帧结构，编写程序来解析并显示帧的各个字段，并将获得的数据字段组合写入输出文件。Ethernet 帧数据从输入文件中获得，默认的输入文件为二进制数据文件。在本练习中为了简便起见，只读取帧校验字段值而不进行 CRC 校验，每个帧的数据字段按最大长度 100B 封装。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 FrameParse.exe，则程序的命令行

格式为：

```
FrameParse input_file output_file
```

其中，input_file 为输入文件，output_file 为输出文件。

(2) 要求将全部字段内容显示在控制台上，具体格式为：

```
帧 1 开始解析
前导码:xx xx xx xx xx xx xx
帧前定界符:xx
目的地址:xx-xx-xx-xx-xx-xx
源地址:xx-xx-xx-xx-xx-xx
长度字段:xx xx
数据字段:…
帧校验字段:xx xx xx xx
帧 2 开始解析
…
```

由于帧数据字段封装的是文本信息，因此该字段内容请按字符串格式输出，其他各字段均按十六进制格式输出。

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

3.3.2 关键问题

1. 文件的读写操作

由于 Ethernet 帧数据需要从输入文件获得，而数据字段内容也需要写入输出文件，因此，首先需要完成对文件的相关操作，例如文件的打开、读取、写入与定位等。通过 fstream、ifstream 与 ofstream，可以完成相关的文件操作。其中，fstream 可以对文件进行读写操作；ifstream 可以对文件进行读操作；ofstream 可以对文件进行写操作。

以下是常用的文件操作函数。

- file.open()：按指定方式打开文件，例如十进制、二进制或十六进制。
- file.read()：从指针位置读取指定字节的数据。
- file.write()：向指针位置写入指定字节的数据。
- file.get()：从指针位置读取 1B 的数据。
- file.put()：向指针位置写入 1B 的数据。
- file.seekg()：将指针移到指定位置。
- file.tellg()：获得指针位置的偏移量。

下面给出获得文件数据长度的伪代码：

```
//打开指定输入文件
```

```
fstream file;
file.open(argv[1],ios::binary);
//将指针移到文件结尾
file.seekg(0,ios::end);
int length=file.tellg();
```

2. 解析帧头部各个字段

在完成 Ethernet 帧解析的过程中，首先需要解析的是帧头部的各个字段。这时，只需将前导码(7B)、帧前定界符(1B)、目的地址(6B)、源地址(6B)、长度字段(2B)，根据每个字段的规定长度依次读取，然后按照题目要求的格式输出。在对每个字段的值进行读取的过程中，需要注意该字段值的存取次序(顺序或逆序)，这就涉及网络字节序的问题。

由于前导码与帧前定界符共同起接收同步作用，前导码的值为 7 个 10101010(十六进制为 0xaa)，帧前定界符的值为 10101011(十六进制为 0xab)，因此 aaaaaaaaaaaaaaab 代表着一个帧的开始。由于在输入文件中有可能封装多个帧，因此需要通过前导码与帧前定界符找到每个帧的开始位置。在本练习中为了简便起见，不考虑在数据字段中出现与前导码、帧前定界符相同值的情况。

下面给出确定帧开始位置的伪代码：

```
//将指针移到文件开始
file.seekg(0,ios::beg);
while(文件未结束)
{
    //查找前导码位置
    for(i=0;i<7;i++)
        if(file.get()!=0xaa)
            …
    //查找帧前定界符位置
    if(file.get()!=0xab)
        …
}
```

3. 解析数据字段

在进行数据字段的解析过程中，需要注意数据字段的长度问题。IEEE 802.3 标准规定数据字段的最小长度为 46B，最大长度为 1500B。如果数据长度小于 46B，通过填充“0”来补足 46B，但是这些“0”的个数不计入长度字段。根据长度字段值来处理数据字段，如果得到长度字段值小于 46，则需要将填充“0”从数据字段去掉；如果得到长度字段值等于 1500，则需要判断其后是否仍有一个帧。

下面给出输出数据字段的伪代码：

```
//数据从缓冲区写入输出文件
char* data=new char[length];
file.read(data,length);
outfile.write(data,length);
```

```
//数据长度小于 46B,去掉填充数据
if(length<46)
    for(i=0;i<46-length;i++)
        file.get();
```

4. 程序流程图

图3-4给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要有一个输入文件名与一个输出文件名。如果命令行参数的个数不是两个,那么程序在输出错误信息后退出。在主程序流程中,需要判断输入文件能否打开、帧同步信息是否存在、长度字段值是否小于46,以及是否还有帧需要解析。

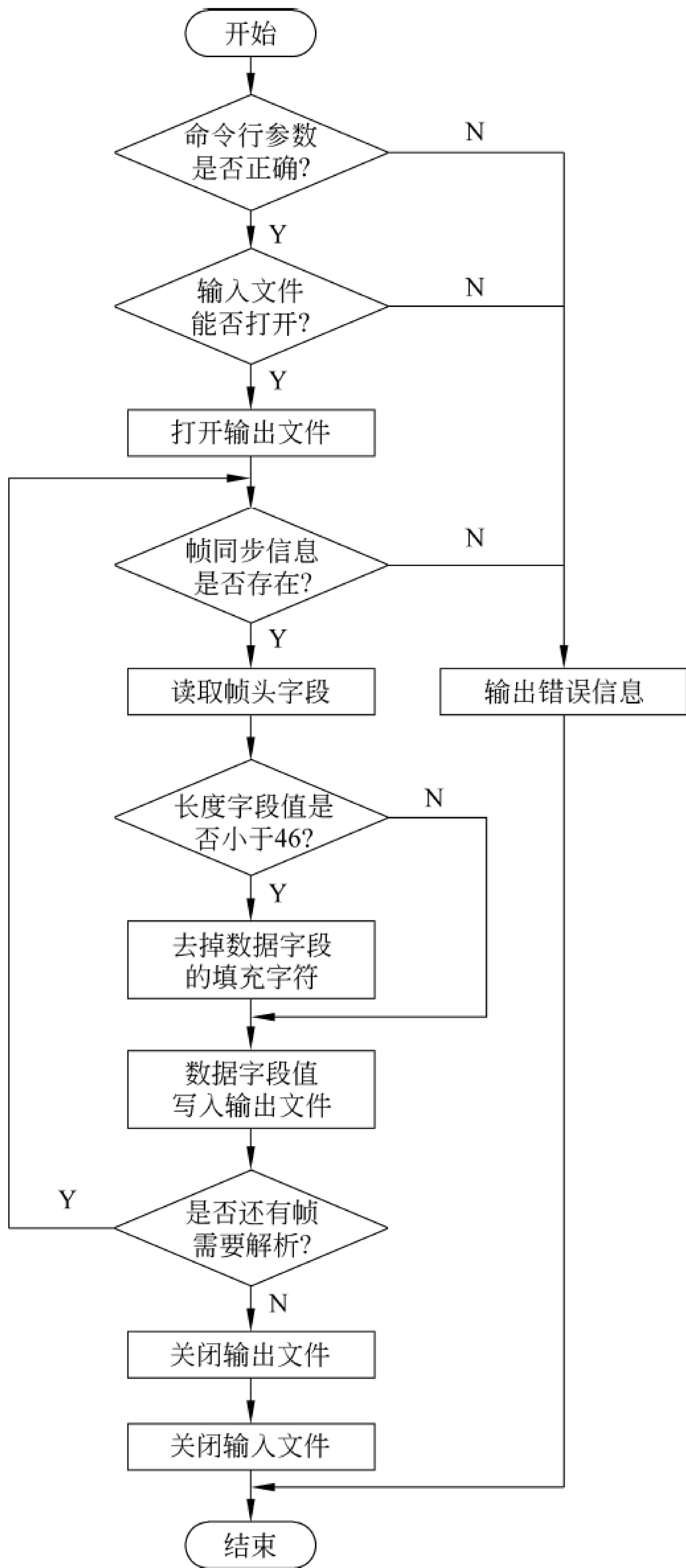


图3-4　主程序流程图

3.3.3 程序源代码

下面给出 Ethernet 帧解析程序的源代码：

```
//FrameParse.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "fstream"
#include "iostream"
using namespace std;

void main(int argc, void* argv[])
{
    if(argc!=3)                                          //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:FrameParse input_file
            output_file"<<endl;
        return;
    }

    fstream outfile;                                     //创建输出文件流
    outfile.open(argv[2],ios::out|ios::binary);          //打开输出文件

    fstream infile;                                      //创建输入文件流
    infile.open(argv[1],ios::in|ios::binary);            //打开输入文件
    if(!infile.is_open())                                //输入文件能否打开
    {
        cout<<"无法打开输入文件"<<endl;
        return;
    }

    bool bframe=true;
    int nframes=0;
    int nframenum=0;
    int nframelen=0;
    while(bframe)                                        //判断帧是否需拆分
    {
        int i=0;
        nframenum++;
        cout<<endl<<"帧"<<nframenum<<"开始解析"<<endl;

        nframes=infile.tellg();
        for(int i=0;i<7;i++)                             //查找 7B 前导码
```

```
            {
                if(infile.get()!=0xaa)
                {
                    cout<<"没找到合法的帧"<<endl;
                    infile.close();
                    return;
                }
            }

            if(infile.get()!=0xab)                          //查找 1B 帧前定界符
            {
                cout<<"没找到合法的帧"<<endl;
                infile.close();
                return;
            }

            infile.seekg(nframes,ios::beg);
            cout<<endl<<"前导码:";
            for(int i=0;i<7;i++)
                cout<<hex<<infile.get()<<dec<<" ";          //写入 7B 前导码
            cout<<endl<<"帧前定界符:";
            cout<<hex<<infile.get();                        //写入 1B 帧前定界符

            cout<<endl<<"目的地址:";
            for(int i=0;i<6;i++)
            {
                cout<<hex<<infile.get()<<dec;               //读取 6B 目的地址
                if(i!=5)                                    //格式:xx-xx-xx-xx-xx-xx
                    cout<<"-";
            }

            cout<<endl<<"源地址:";
            for(int i=0;i<6;i++)
            {
                cout<<hex<<infile.get()<<dec;               //读取 6B 源地址
                if(i!=5)                                    //格式:xx-xx-xx-xx-xx-xx
                    cout<<"-";
            }

            cout<<endl<<"长度字段:";
            cout<<hex<<infile.get()<<" ";                   //读取长度字段的高 8 位
            nframelen=infile.get();                         //读取长度字段的低 8 位
            cout<<hex<<nframelen;

            char* data=new char[nframelen];
            infile.read(data,nframelen);                    //数据从输入文件读到 data
            outfile.write(data,nframelen);                  //数据从 data 写入输出文件
            cout<<endl<<"数据字段:";
            for(i=0;i<nframelen;i++)
```

```
            cout<<data[i];
        delete data;

        if(nframelen<100)                         //若长度字段值大于 100,则分为多帧
            bframe=false;
        if(nframelen<46)                          //若长度字段值小于 46,则补 0
        {
            for(i=0;i<46-nframelen;i++)
                infile.get();
        }

        cout<<endl<<"帧校验字段:";
        for(i=0;i<4;i++)
            cout<<hex<<infile.get()<<dec<<" ";    //写入 4B 帧校验
        cout<<endl;
    }

    cout<<endl<<"帧全部解析完成"<<endl;
    outfile.close();                              //关闭输出文件
    infile.close();                               //关闭输入文件

    return;
}
```

图 3-5 给出了 Ethernet 帧的解析过程。程序命令行输入为 FrameParse input output。程序从 input 读取数据,并根据长度进行拆分、填充等操作,然后依次将前导码、帧前定界符、目的地址、源地址、长度、数据与帧校验字段的值写入 output。

```
命令提示符
E:\Test\FrameParse\Debug>FrameParse

请按以下格式输入命令行: FrameParse input_file output_file

E:\Test\FrameParse\Debug>FrameParse input output

帧1开始解析

前导码: aa aa aa aa aa aa aa
帧前定界符: ab
目的地址: 0-0-e4-86-3a-dc
源地址: 0-0-80-1a-e6-65
长度字段: 0 64
数据字段: Nankai University was founded in 1919 by the famous patriotic educator
s in Chinese modern history, M
帧校验字段: ff ff ff ff

帧2开始解析

前导码: aa aa aa aa aa aa aa
帧前定界符: ab
目的地址: 0-0-e4-86-3a-dc
源地址: 0-0-80-1a-e6-65
长度字段: 0 20
数据字段: r. Zhang Boling and Mr. Yan Xiu.
帧校验字段: ff ff ff ff

帧全部解析完成
```

图 3-5 Ethernet 帧的解析过程

3.4 练　习　题

根据IEEE 802.3格式的Ethernet帧结构,编写程序将原始数据封装成一个或多个帧,并将这些帧的各个字段值写入输出文件。原始数据从输入文件中获得,默认的输入文件为二进制数据文件。在本练习中为了简便起见,只填写帧校验字段而不进行CRC校验,每个帧的数据字段按最大长度100B封装。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为FrameEncap.exe,则程序的命令行格式为:

```
FrameEncap input_file output_file
```

其中,input_file为输入文件,output_file为输出文件。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
帧 1 开始封装
长度字段:xx xx
数据字段:…
帧 2 开始封装
…
```

由于帧数据字段封装的是文本信息,因此该字段内容请按字符串格式输出,其他各字段均按十六进制格式输出。

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第4章 Ethernet帧的CRC校验

4.1 设计目的

网络中的数据传输最终要通过物理线路完成，但是物理层无法保证数据传输不出错，因此在物理层之上设计了数据链路层。数据链路层的作用是在原始的、有差错的物理线路上，采用差错控制、流量控制等方法，将有差错的物理线路改造成无差错的数据链路，向上面的网络层提供高质量的传输服务。本章练习的目的是通过 Ethernet 帧的 CRC 校验，了解网络协议中校验和的计算过程与作用。

4.2 相关知识

本章涉及的相关知识包括 CRC 校验的概念与工作原理。

4.2.1 CRC 校验的概念

数据链路层的差错控制方法主要分为两种：纠错码与检错码。其中，纠错码需要能够发现并自动纠正传输差错，要求它必须为数据添加足够的冗余信息，这就造成它的实现困难与应用不够广泛。检错码只需要带有一定数量的冗余信息，使接收方能够发现传输差错并要求发送方重传数据。目前，常见的检错码主要有奇偶校验码与循环冗余编码(Cyclic Redundancy Code，CRC)。

CRC 校验是应用最广泛的检错码方法之一，它具有检错能力强且容易实现的特点。CRC 校验在通信领域广泛用于实现差错控制。CRC 校验过程可以简单描述为：在发送端，根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的 r 位二进制码序列(CRC 码)，附在原始信息后构成一个 $k+r$ 位的二进制码序列(信息码)，然后发送出去。在接收端，根据信息码和 CRC 码之间所遵循的规则进行检验，以确定在传输过程中是否出错。在差错控制理论中，这个规则称为“生成多项式”。

在代数编码理论中，可将一个码组表示为一个多项式，码组中的各个码元作为多项式的系数。例如，100000111 可以表示为 $1 \cdot x^8 + 0 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$，即 $x^8 + x^2 + x + 1$。CRC 生成多项式 $G(x)$ 由不同的协议来定义，目前已有多种生成多项式被列入国际标准中。$G(x)$ 的结构及检错效果经过严格的数学分析与实

验验证。表 4-1 给出了几种 CRC 标准的资料。

表 4-1　CRC 标准的资料

名　称	生成多项式 $G(x)$	应 用 实 例
CRC-4	x^4+x+1	ITU G.704
CRC-8	x^8+x^2+x+1	
CRC-12	$x^{12}+x^{11}+x^3+x+1$	
CRC-16	$x^{16}+x^{12}+x^2+1$	IBM SDLC
CRC-ITU	$x^{16}+x^{12}+x^5+1$	ISO HDLC、ITU X.25、V.34/V.41/V.42 等
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	ZIP、RAR、IEEE 802.3、IEEE 802.6、IEEE 1394、PPP-FCS 等

CRC 校验的工作过程可以描述如下。

(1) 发送端的发送数据多项式为 $F(x)\cdot x^k$，其中 k 为生成多项式的最高幂的值。例如，CRC-8 的最高幂值为 8，则发送 $F(x)\cdot x^8$。对于二进制乘法来说，$F(x)\cdot x^8$ 的意义是将发送数据比特序列左移 8 位用来放入余数。

(2) 将 $F(x)\cdot x^k$ 除以生成多项式 $G(x)$，得

$$\frac{F(x)\cdot x^k}{G(x)}=Q(x)+\frac{R(x)}{G(x)}$$

式中 $R(x)$ 为余数多项式。

(3) 将 $F(x)\cdot x^k+R(x)$ 作为整体，从发送端通过通信信道发送到接收端。

(4) 接收端对接收数据多项式 $F'(x)$ 采用同样的运算，即

$$\frac{F'(x)\cdot x^k}{G(x)}=Q(x)+\frac{R'(x)}{G(x)}$$

求得余数多项式 $R'(x)$。

(5) 将计算出的余数多项式 $R'(x)$ 与接收余数多项式 $R(x)$ 比较，以此来判断数据在传输过程中是否出错。

4.2.2　CRC 校验的例子

实际的 CRC 校验采用二进制模二算法(减法不错位、加法不进位)，这是一种异或操作。下面通过实例进一步说明 CRC 校验过程。

(1) 发送数据 $F(x)$ 为 10010010(8 位)。

(2) 生成多项式 $G(x)$ 为 100000111(9 位，$k=8$)。

(3) 将发送数据 $F(x)$ 乘以 2^8，产生的乘积应为 1001001000000000。

(4) 将这个乘积按模二除法除以生成多项式 $G(x)$：

```
                                 10010001 ←— Q(x)
G(x) —→ 100000111 √1001001000000000 ←— F(x) · x^k
                   100000111
                   ---------
                      100011000
                      100000111
                      ---------
                          111110000
                          100000111
                          ---------
                           11110111 ←— R(x)
```

求得余数多项式 $R(x)$为 11110111。

(5) 将余数多项式 $R(x)$加到乘积中,得

```
10010010      11110111
└──────┘      └──────┘
发送数据    CRC 校验码
└──────────────────────┘
      带 CRC 校验码
        的发送数据
```

(6) 如果数据在传输过程中没有出错,接收端收到的带有 CRC 码的数据一定能被相同的生成多项式 $R(x)$整除,即

```
                                 10010001 ←— Q(x)
G(x) —→ 100000111 √1001001011110111 ←— F'(x)
                   100000111
                   ---------
                      100010111
                      100000111
                      ---------
                          100000111
                          100000111
                          ---------
                                  0
```

4.2.3 CRC 校验的硬件实现

在实际的网络应用中,CRC 校验过程可以通过硬件或软件来实现。目前,很多超大规模集成电路芯片可以实现 CRC 校验。

图 4-1 给出了 CRC 运算的通用电路图。CRC 校验的生成多项式 $G(x)$通常用 n 次多项式来定义:$G(x)=x^n+g_{n-1}\cdot x^{n+1}+\cdots+g_i\cdot x^i+\cdots+g_2\cdot x^2+g_1\cdot x+1$,其中 g_i 为 0 或 1。通常,CRC 校验可用有 n 个存储器级的移位寄存器来实现。如果多项式相应项的系数为 1,则相应的存储器级输入端的模 2 加法器有分支。在系统开始工作之前,所有移位寄存器级全部置 0 或 1。图 4-1 中的输入端输入的是原始数据序列,移位寄存器各级输出 b_0、b_1、…、b_{n-2}、b_{n-1}是 CRC 码字。其中,b_0 和 b_{n-1} 分别是最低有效位和最高有效位。

图 4-2 给出了 CRC 运算的实现方法。以 CRC-8 校验($x^8+x^2+x^1+1$)为例,它由多个移位寄存器和加法器组成。在编码与解码前先将各寄存器初始化为 0,输入位作为最右边异或操作的输入之一。三个寄存器上的移位操作同时进行(均为左移 1 位),左边寄存器的最左边的位作为三个异或操作的输入之一。每次进行移位操作时,最右边的寄存器作为中

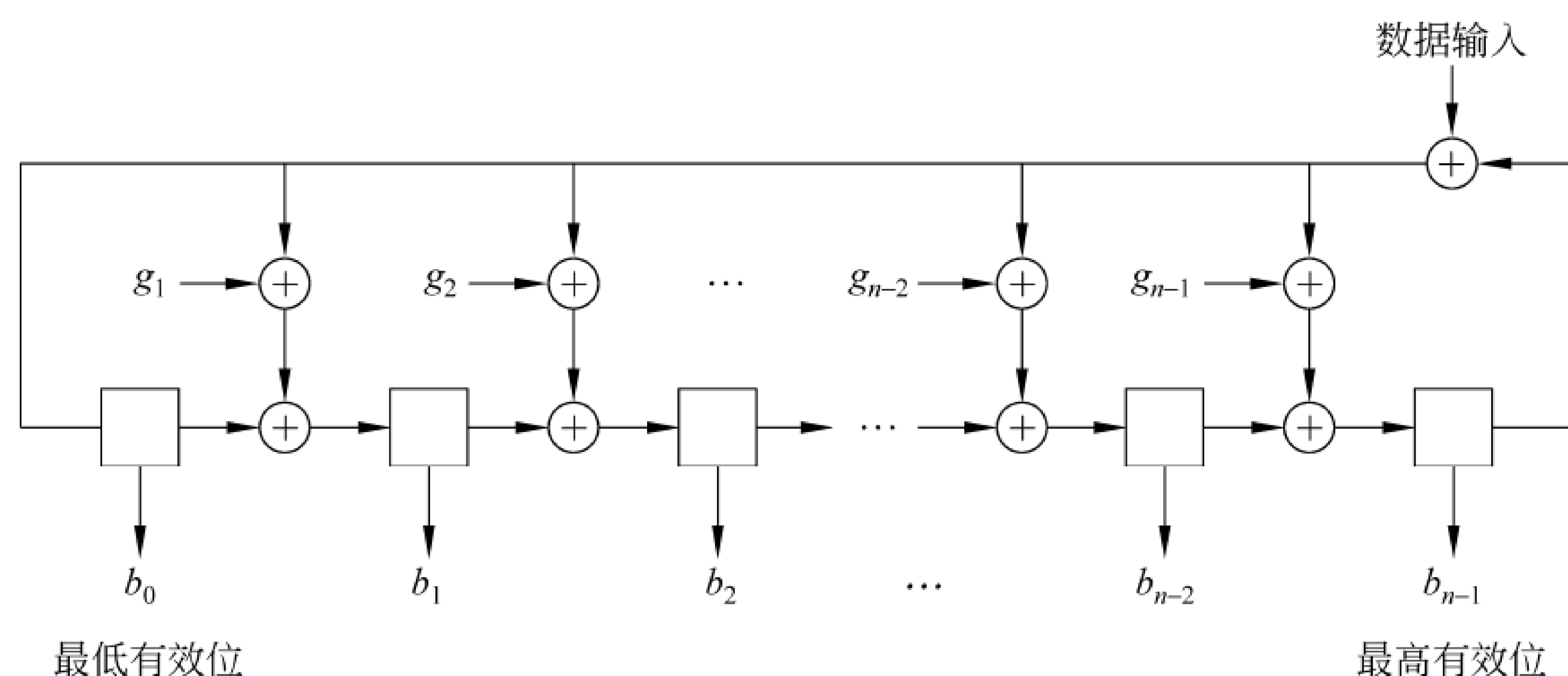


图 4-1　CRC 运算的通用电路图

间的异或操作的输入之一，中间的寄存器作为最左边异或操作的输入之一，各个异或操作的结果作为左边寄存器的移入位。重复以上步骤，每输入 1 位就进行一次移位操作，直到输入所有需要计算的数据为止。这时，寄存器组中的数据就是 CRC-8 运算的结果。

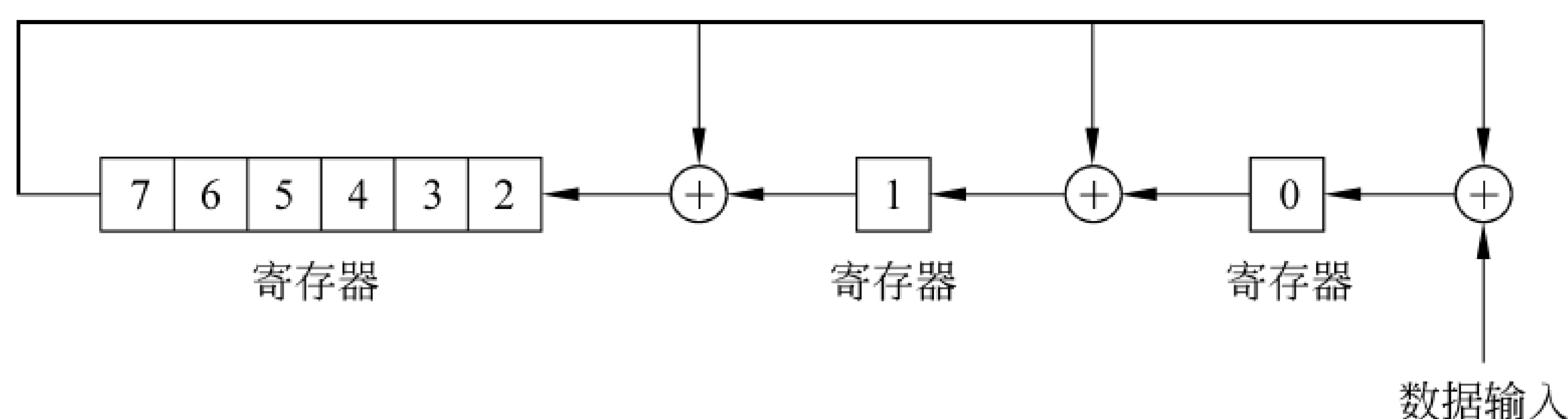


图 4-2　CRC 运算的实现方法

4.2.4　CRC 校验的主要特点

CRC 校验码的检错能力很强，它除了能够检查出离散错之外，还能检查出突发错。CRC 校验码具有以下检错能力。

- CRC 校验码可以检测出所有单个错。
- CRC 校验码可以检测出所有奇数位错。
- CRC 校验码可以检测出所有双比特的错。
- CRC 校验码可以检测出所有小于或等于校验位长度的突发错。
- CRC 校验码可以检测出长度为 $(k+1)$ 位突发错的概率为 $[1-(1/2)^{k-1}]$。

例如，如果 $k=8$，则 CRC 校验码可以检测出所有小于或等于 8 位的突发错，并且能以 99.218% 的概率检测出长度为 9 位的突发错。

4.3　例题分析

4.3.1　设计要求

根据 IEEE 802.3 标准的 Ethernet 帧结构，编写程序来封装一个帧且进行 CRC 校验，并将帧的各个字段值写入输出文件。在本练习中为了简便起见，CRC 校验采用 8 位的

CRC-8 校验,原始数据内容为"Hello world!"。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 CrcEncode.exe,则程序的命令行格式为:

```
CrcEncode output_file
```

其中,output_file 为输出文件。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
长度字段:xx
数据字段:…
帧校验字段:xx
```

由于帧数据字段封装的是文本信息,因此该字段内容请按字符串格式输出,其他各字段均按十六进制格式输出。

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

4.3.2 关键问题

1. CRC 校验的过程

由于本题采用的是简单的 CRC-8 校验,因此帧校验字段的长度为 1B。下面,举例说明本程序如何实现 CRC-8 校验算法。假设数据为 10001010,CRC-8 校验的生成多项式为 $G(x)=x^8+x^2+x+1$,即 100000111。图 4-3 给出了 CRC-8 校验的工作过程。在源节点,将数据 10001010 后面补 8 个 0,除以生成多项式 100000111,得到余数 10111111;在目的节点,将数据 10001010 后面添加余数 10111111,除以生成多项式 100000111,如果得到余数 0,则说明 CRC 校验正确。

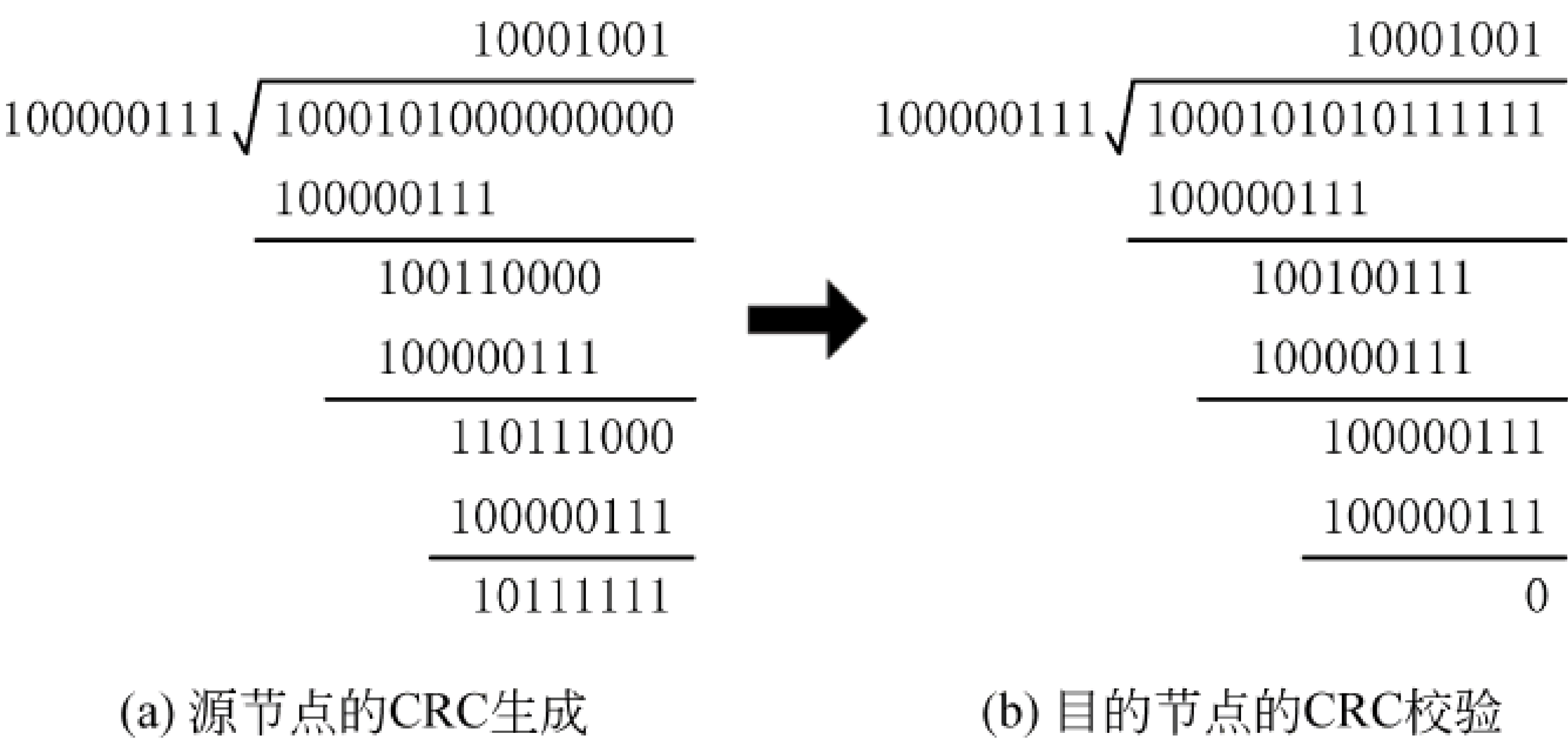


图 4-3 CRC-8 校验的工作过程

CRC 编码实际上是一个循环移位的模 2 运算。对于 CRC-8 校验,假设 CRC 是一个 9 位的寄存器,通过反复进行移位与模 2 运算操作,最终该寄存器中的值去掉最高位即为余数。图 4-4 给出了 CRC-8 校验的程序流程图。寄存器中的后 8 位是经过 CRC-8 校验的余

数,这样只需要使用寄存器中的后8位,因此可以将9位寄存器简化为8位。

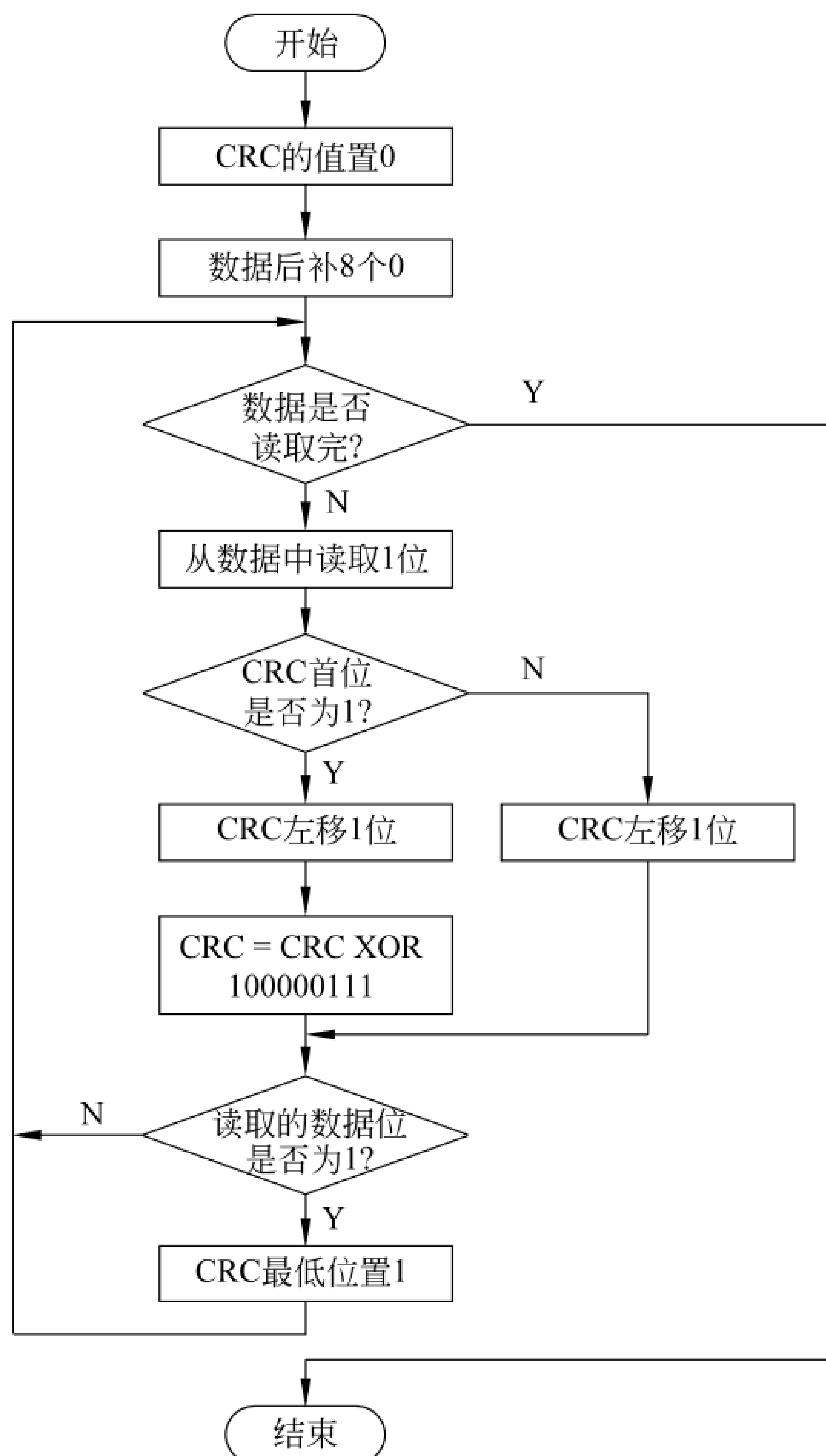


图4-4　CRC-8校验的程序流程图

首先需要构造一个8位的寄存器CRC,并将CRC的初始值设置为0,然后将数据依次移入CRC的最低位,同时将CRC的最高位移出。当移出的数据为1时,CRC才和00000111进行异或(XOR)运算;当移出的数据为0时,不做任何运算。同时,每次CRC中的数据左移后,需要从输入数据中读入1位新的数据。由于左移时CRC的最低位补0,因此当读入的数据最高位为1时,还需要将CRC的最低位置1。

下面给出CRC-8校验的伪代码:

```
//CRC是一个8位的寄存器
将 CRC 的值置为 0
在原始数据 input 后添加 8 个 0
while(数据未处理完)
{
    if(CRC 首位为 1)
        CRC 左移 1 位
```

```
        CRC=CRC XOR 00000111
    else
        CRC 左移 1 位
    if(从 input 中读 1 位新的数据为 1)
        将 CRC 的最低位置 1
}
```

2. CRC 校验的范围

根据 IEEE 802.3 标准的规定，Ethernet 帧校验的范围包括目的地址、源地址、长度与数据字段。也就是说，当源节点对某个帧进行 CRC 校验时，需要对从“目的地址”开始到“数据”字段的数据依次进行计算，然后将得到的余数填入帧校验字段中；当目的节点对该帧进行 CRC 校验时，需要对从“目的地址”开始到“帧校验”字段的数据依次进行计算，如果最终得到的余数为 0，则说明 CRC 校验正确。

下面给出确定 CRC 校验范围的伪代码：

```
nCrcS=目的地址位置
数据写入文件
nCrc=帧校验字段位置
total=CRC 校验的数据长度
将文件指针指向 nCrcS
while(数据未处理完)
    CRC 校验过程
将 CRC 值写入帧校验字段
```

3. 程序流程图

图 4-5 给出了主程序流程图。要求输入的命令行参数必须正确，除了程序本身的名称以外，还需要一个输入文件名作为参数。如果命令行参数的个数不是一个，则程序在输出错误信息后退出。

4.3.3 程序源代码

下面给出 Ethernet 帧校验程序的源代码：

```
//CrcEncode.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "fstream"
#include "iostream"
using namespace std;

void main(int argc, char* argv[])
{
    if(argc!=2)                                    //检查命令行参数
```

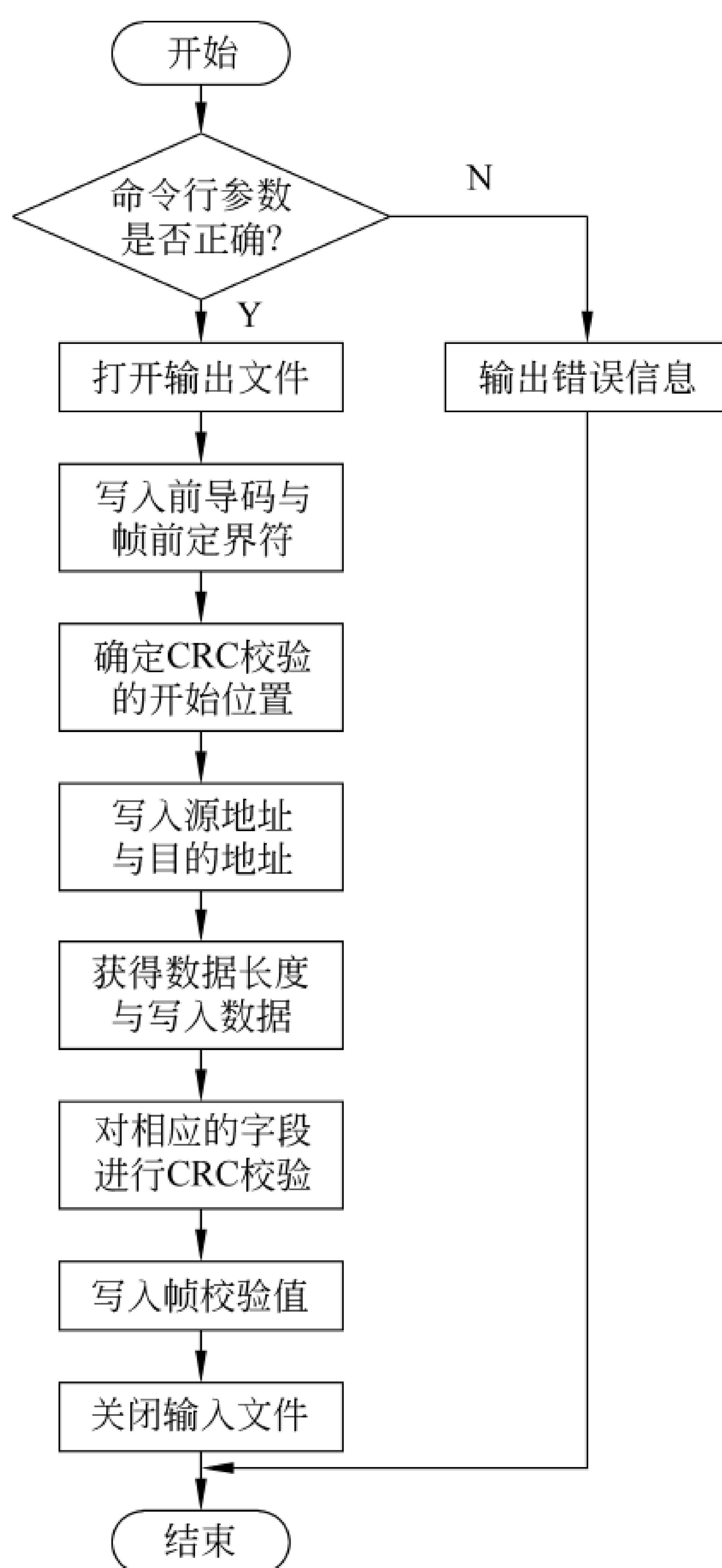


图 4-5　主程序流程图

```
{
    cout<<endl<<"请按以下格式输入命令行:CrcEncode output_file"<<endl;
    return;
}

fstream outfile;                                        //创建输出文件流
outfile.open(argv[1],ios::out|ios::binary);             //打开输出文件
for(int i=0;i<7;i++)
    outfile.put(char(0xaa));                            //写入 7B 前导码
outfile.put(char(0xab));                                //写入 1B 帧前定界符

int nCrcS=int(outfile.tellp());                         //开始计算 CRC 位置
char dst_addr[6]={char(0x00),char(0x00),char(0xe4),char(0x86),char(0x3a), char
(0xdc)};
outfile.write(dst_addr,sizeof(dst_addr));               //写入 6B 目的地址
char src_addr[6]={char(0x00),char(0x00),char(0x80),char(0x1a),char(0xe6), char
(0x65)};
```

```
    outfile.write(src_addr,sizeof(src_addr));           //写入 6B 源地址

    char data[]="Hello world!";
    int length=strlen(data);                            //获得数据长度
    outfile.put(char(length/256));                      //写入 2B 长度字段
    outfile.put(char(length%256));
    cout<<endl<<"长度字段:"<<hex<<length<<dec<<"("<<length<<")"<<endl;

    outfile.write(data,length);                         //data 写入数据字段
    cout<<"数据字段:"<<data<<endl;

    int nCrc=int(outfile.tellp());                      //获得帧校验字段位置
    outfile.put(char(0x00));                            //数据后补 1B 的 0,CRC 计算
    int total=int(outfile.tellp())-nCrcS;               //获得 CRC 计算的数据长度
    outfile.seekg(nCrcS,ios::beg);                      //文件指针指向目的地址
    unsigned char crc=0;                                //初始化 CRC 余数为 0
    while(total--)
    {
        char temp;
        outfile.get(temp);                              //读取 1B 的数据

        //模拟数据除以 100000111 的二进制除法过程
        for(unsigned char i=(unsigned char)0x80;i>0;i>>=1)
        {
            if(crc&0x80)                                //余数最高位为 1,除法运算
            {
                crc<<=1;                                //左移 1 位
                if(unsigned char(temp)&i)
                    crc^=0x01;                          //相应位递补余数末位
                crc^=0x07;                              //除法运算(减去除数低 8 位)
            }
            else
            {
                crc<<=1;                                //左移 1 位
                if(unsigned char(temp)&i)
                    crc^=0x01;                          //相应位递补余数末位
            }
        }
    }

    outfile.seekp(nCrc,ios::beg);                       //文件指针移到数据字段
    outfile.put(crc);                                   //余数写入帧校验字段
    cout<<"帧校验字段:"<<hex<<(int)crc<<dec<<"("<<(int)crc<<")"<<endl;
```

```
    cout<<endl<<"帧封装与 CRC 校验完成"<<endl;
    outfile.close();

    return;
}
```

图 4-6 给出了 Ethernet 帧校验的过程。程序命令行输入为 CrcEncode output。程序将除了帧校验之外的其他字段依次写入 output，对目的地址、源地址、长度与数据部分进行 CRC 计算，然后将获得的帧校验数值写入 output。

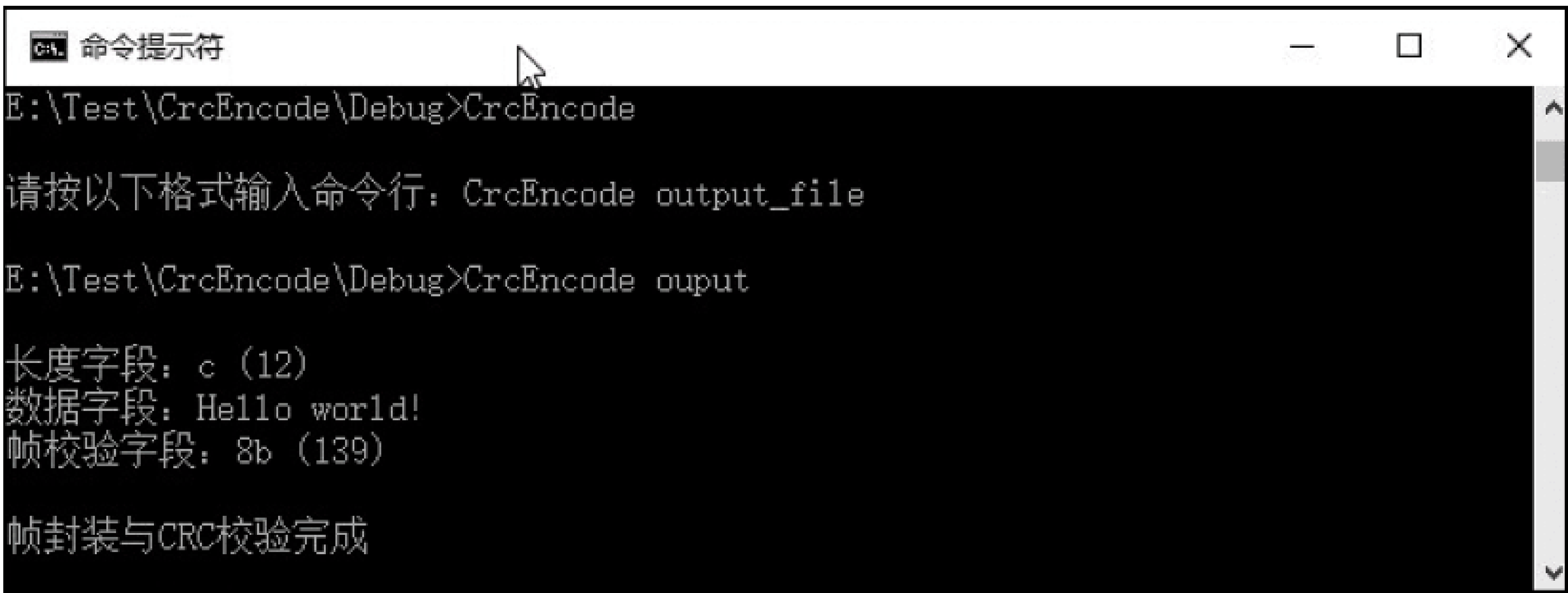


图 4-6　Ethernet 帧校验的过程

4.4　练　习　题

根据 IEEE 802.3 格式的 Ethernet 帧结构，编写程序来解析一个帧且进行 CRC 校验，并判断该帧在传输过程中是否出错。Ethernet 帧数据从输入文件中获得，默认的输入文件为二进制数据文件。在本练习中为了简便起见，CRC 校验采用 8 位的 CRC-8 校验。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 CrcDecode.exe，则程序的命令行格式为：

```
CrcDecode input_file
```

其中的 input_file 为输入文件。

(2) 要求将部分字段内容显示在控制台上，具体格式为：

```
目的地址:xx-xx-xx-xx-xx-xx
源地址:xx-xx-xx-xx-xx-xx
长度字段:xx
数据字段:…
帧校验字段:xx(CRC 校验正确或错误)
```

由于帧数据字段封装的是文本信息，因此该字段内容请按字符串格式输出，其他各字段

均按十六进制格式输出。

（3）要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

（4）要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 5 章 IP 地址的合法性判断

5.1 设 计 目 的

IP 地址是 TCP/IP 协议在网络层使用的地址，它用来唯一地标识一台接入 Internet 的主机。熟悉 IP 地址对于理解网络协议的概念、层次结构与执行过程有很大的帮助。本章练习的目的是根据 IP 地址的基本结构，通过分析 IP 地址来了解地址格式与类型，从而深入理解网络层协议的工作过程。

5.2 相 关 知 识

本章涉及的相关知识包括 IP 地址的概念与分类方法，以及子网的划分方法。

5.2.1 IP 地址的基本概念

Internet 就是一个规模很大的互联网络，其中连接数量众多的计算机与网络设备，支撑它运行的核心技术是 TCP/IP 协议。在 TCP/IP 协议体系中，一个重要的技术特征就表现在：它提供了统一的网络地址分配方案，所有网络设备在 Internet 中都有唯一的 IP 地址。RFC791 文档定义了 IPv4 协议的基本内容，其中指出了三个重要概念：名字(name)、地址(address)与路径(route)。这里，名字说明这个节点是谁，地址说明这个节点在哪里，而路径说明如何找到这个节点。

互联网络是由多个不同类型的网络互联而成，这些网络可以是广域网、城域网或局域网，也可以是以太网、光纤网或无线网。连接到每个网络的每台计算机都有一块网卡。也就是说，每台计算机都有一个 MAC 地址。这个 MAC 地址是一个数据链路层地址，人们通常将它称为“物理地址”。IP 地址是一个网络层的地址，主要用于路由器的寻址，因此 IP 地址采用层次结构。相对于数据链路层的固定不变的物理地址，网络层地址可以由网管人员分配和通过软件来设置，因此人们也将它称为“逻辑地址”。

1981 年，IETF 制订了最初的 IPv4 地址方案，当时的现状是网络规模比较小，用户通常是通过终端来接入 ARPANET。IPv4 地址采用分层结构。图 5-1 给出了 IPv4 地址的结构。IP 地址由两个部分组成：网络号与主机号。其中，网络号用来标

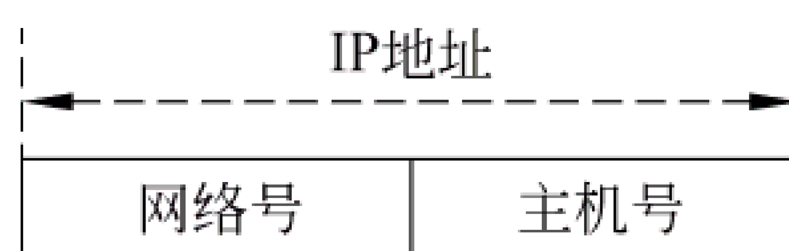


图 5-1 IPv4 地址的结构

识一个网络；主机号用来标识一台主机或路由器。当源主机与目的主机之间通信时，源主机与目的主机的 IP 地址都封装在 IPv4 包中，通过中间的路由器为 IPv4 包选择合适的转发路径。由此可见，连接到 Internet 的每台主机（计算机或路由器）至少有一个 IP 地址。

IPv4 地址的长度为 32 位，采用点分十进制（dotted decimal）表示。通常采用 $X.X.X.X$ 的格式表示，每个 X 是一个 8 位的数字。其中，最小的数字为 00000000，用点分十进制数表示为 $X=0$；最大的数字为 11111111，用点分十进制数表示为 $X=1\times2^7+1\times2^6+1\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0=128+64+32+16+8+4+2+1=255$，因此每个 X 的取值范围为 0～255。RFC791 定义了最初的 IPv4 地址结构。后来，针对 IPv4 地址出现了众多的 RFC 文档，其中涉及地址分配、专用地址、地址转换等内容。

5.2.2 IP 地址的分类方法

在最初的 IPv4 地址结构中，仅采用网络号加主机号的两层结构。在这个阶段，根据地址的取值范围与空间大小，IPv4 地址可以分为 5 种类型：A 类、B 类、C 类、D 类与 E 类。图 5-2 给出了 IPv4 地址的分类。IP 地址中的前 5 位包含地址类型标识，A 类地址的第一位为“0”，B 类地址的前两位为“10”，C 类地址的前三位为“110”，D 类地址的前 4 位为“1110”，E 类地址的前 5 位为“11110”。其中，A 类、B 类与 C 类地址是基本 IP 地址，D 类与 E 类地址主要用于特殊或实验用途。

图 5-2　IPv4 地址的分类

A 类地址的网络号长度为 7 位，主机号长度为 24 位。这样，A 类地址的取值范围为 1.0.0.0～127.255.255.255。由于 A 类地址的网络号长度为 7 位，因此理论上可分配 $2^7-2=126$ 个 A 类网络。由于 A 类地址的主机号长度为 24 位，因此每个 A 类网络可容纳 $2^{24}-2=16\ 777\ 214$ 台主机。显然，A 类地址适用于拥有大量主机的大型网络，但某个组织拥有如此多的主机并不现实，因此其中很多地址实际上是浪费的。

B 类地址的网络号长度为 14 位，主机号长度为 16 位。这样，B 类地址的取值范围为 128.0.0.0～191.255.255.255。由于 B 类地址的网络号长度为 14 位，因此理论上可分配 $2^{14}-2=16\ 382$ 个 B 类网络。由于 B 类地址的主机号长度为 16 位，因此每个 B 类网络可容纳 $2^{16}-2=65\ 534$ 台主机。显然，B 类地址适用于政府组织或大公司，但是某个组织拥有如此多的主机并不多见，因此其中很多地址也是浪费的。

C 类地址的网络号长度为 21 位，主机号长度为 8 位。这样，C 类地址的取值范围为

192.0.0.0～223.255.255.255。由于C类地址的网络号长度为21位，因此理论上可分配$2^{21}-2=2\ 097\ 150$个C类网络。由于C类地址的主机号长度为8位，因此每个C类网络可容纳$2^8-2=254$台主机。显然，C类地址适用于科研机构或小公司，这类网络是需求最大与使用最广泛的。

对于剩余两类地址，D类地址的取值范围为224.0.0.0～239.255.255.255。D类地址不用于标识网络，主要用于特殊用途（如多播地址）。E类地址的取值范围为240.0.0.0～247.255.255.255。E类地址暂时保留，主要用于实验与未来用途。

5.2.3 其他IP地址类型

1. 特殊IP地址

直接广播地址（directed broadcasting address）是A类、B类与C类地址中主机号为全1的地址。图5-3给出了直接广播地址的结构。直接广播地址用来将IP包以广播形式发送给特定网络中的所有主机。直接广播地址只能作为IP包中的目的地址。例如，IP包中的目的地址（201.1.16.255）的主机号为全1，则路由器将它广播给网络（201.1.16.0）中的所有主机。



图5-3 直接广播地址的结构

受限广播地址（limited broadcasting address）是网络号与主机号均为全1的地址，即255.255.255.255。图5-4给出了受限广播地址的结构。受限广播地址用来将IP包以广播形式发送给本地网络中的所有主机。例如，IP包中的目的地址（255.255.255.255）为全1，则路由器将它广播给本地网络中的所有主机。

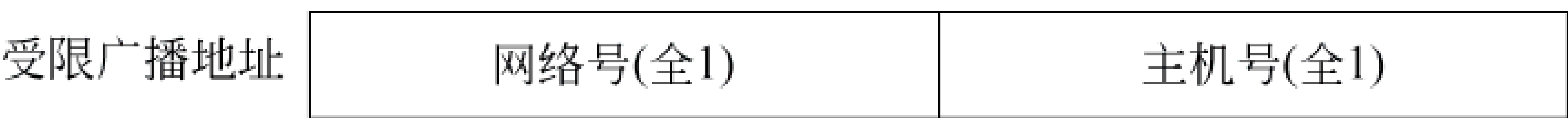


图5-4 受限广播地址的结构

“本网络中的特定主机”地址是A类、B类与C类地址中网络号为全0的地址。图5-5给出了“本网络中的特定主机”地址。“本网络中的特定主机”地址用来将IP包发送给本地网络中的特定主机。例如，IP包中的目的地址（192.0.0.254）的网络号为全0，则路由器将它发送给本网络中的一台特定主机。

回送地址（lookback address）是A类地址中网络号为全1、主机号为全0的地址，即127.0.0.0。图5-6给出了回送地址的结构。回送地址用于网络软件测试与本地进程之间通信。例如，IP包中的目的地址为127.0.0.0，主机与路由器都不会将该IP包转发到公网，而是将它回送给发送该IP包的主机。



图 5-5 “本网络中的特定主机”地址



图 5-6 回送地址的结构

2. 专用 IP 地址

RFC1918 提出在 A 类、B 类与 C 类地址中，各保留一部分地址作为专用的 IP 地址。专用地址用于不连接 Internet 的内部网络。表 5-1 给出了保留的专用地址。内部网络的主机向 Internet 发送 IP 包时，需要将专用地址转换成全局 IP 地址。

表 5-1 保留的专用地址

类　　型	网 络 号	总　　数
A 类	10.1	1
B 类	172.16～172.31	16
C 类	192.168.0～192.168.255	256

理解专用地址需要注意以下几个问题。

- 如果 IP 包中的地址使用 10.1.0.1、172.16.1.12 或 192.168.0.2，则路由器认为这是一个内部网络使用的 IP 地址，不会向 Internet 转发该 IP 包。
- 如果一个组织出于安全等原因，希望组建一个专用的内部网络，不准备连接到 Internet，或者在转发 IP 包到 Internet 时希望使用网络地址转换(NAT)，则该组织就可以使用专用 IP 地址。

5.2.4 IP 地址技术发展

由于初期的 ARPANET 是一个研究性的网络，即使将美国约 2000 所大学与一些研究机构以及其他国家的一些大学接入 ARPANET，总数也不会超过 16 000 个。A 类、B 类与 C 类地址的总数在当时是足够分配的。IPv4 设计者当初没有预见到 Internet 发展得如此之快。1987 年，有人预言：Internet 的主机数量可能增加到 10 万个。当时，大多数专家都不相信这个预测，但是在 1996 年第 10 万台计算机已接入 Internet。2011 年 3 月，最后 5 块 IPv4 地址分配之后，世界上已经没有新的 IPv4 地址可以分配。

1. 子网地址划分

标准分类的 IP 地址存在两个主要问题：①IP 地址的有效利用率；②路由器的工作效率。为了解决这两个问题，人们提出子网(subnet)的概念。RFC940 说明了子网的概念与划分标准。研究子网划分的基本思想是：借用主机号的一部分作为子网号，划分出更多的

子网 IP 地址,而对外部路由器的寻址没有影响。图 5-7 给出了划分子网的 IP 地址结构。划分子网技术的要点是:同一子网中的所有主机使用相同的网络号与子网号;子网的概念可用于 A 类、B 类或 C 类地址;子网之间的距离必须很近,这主要是从路由器工作效率角度考虑的。



图 5-7　划分子网的 IP 地址结构

为了从划分子网的 IP 地址中提取子网号,人们提出了子网掩码(subnet mask)的概念。子网掩码的表示方法为:网络号与子网号置 1,主机号置 0。例如,B 类地址的网络号为 16 位,主机号为 16 位,则可将主机号的前 8 位作为子网号,这时子网掩码用点分十进制表示为 255.255.255.0。同样,子网掩码也适用于未划分子网的 IP 地址,只是将对应的网络号置 1、主机号置 0,则 A 类地址的子网掩码为 255.0.0.0,B 类地址的子网掩码为 255.255.0.0,C 类地址的子网掩码为 255.255.255.0。

子网掩码计算是从 IP 地址中提取子网号的过程。无论是否划分子网,都可以进行子网掩码计算。子网掩码计算是将二进制的 IP 地址与掩码按位进行“与”运算。图 5-8 给出了子网掩码的计算过程。例如,IP 地址为 142.16.2.21,对于未划分子网的情况,掩码为 255.255.0.0,计算后的 IP 地址为 142.16.0.0;对于划分子网的情况,掩码为 255.255.255.0,计算后的 IP 地址为 142.16.2.0,则其网络号为 142.16、子网号为 2。有时,划分子网时需考虑子网号长度不同的情况。RFC1009 文档说明了变长子网的划分。

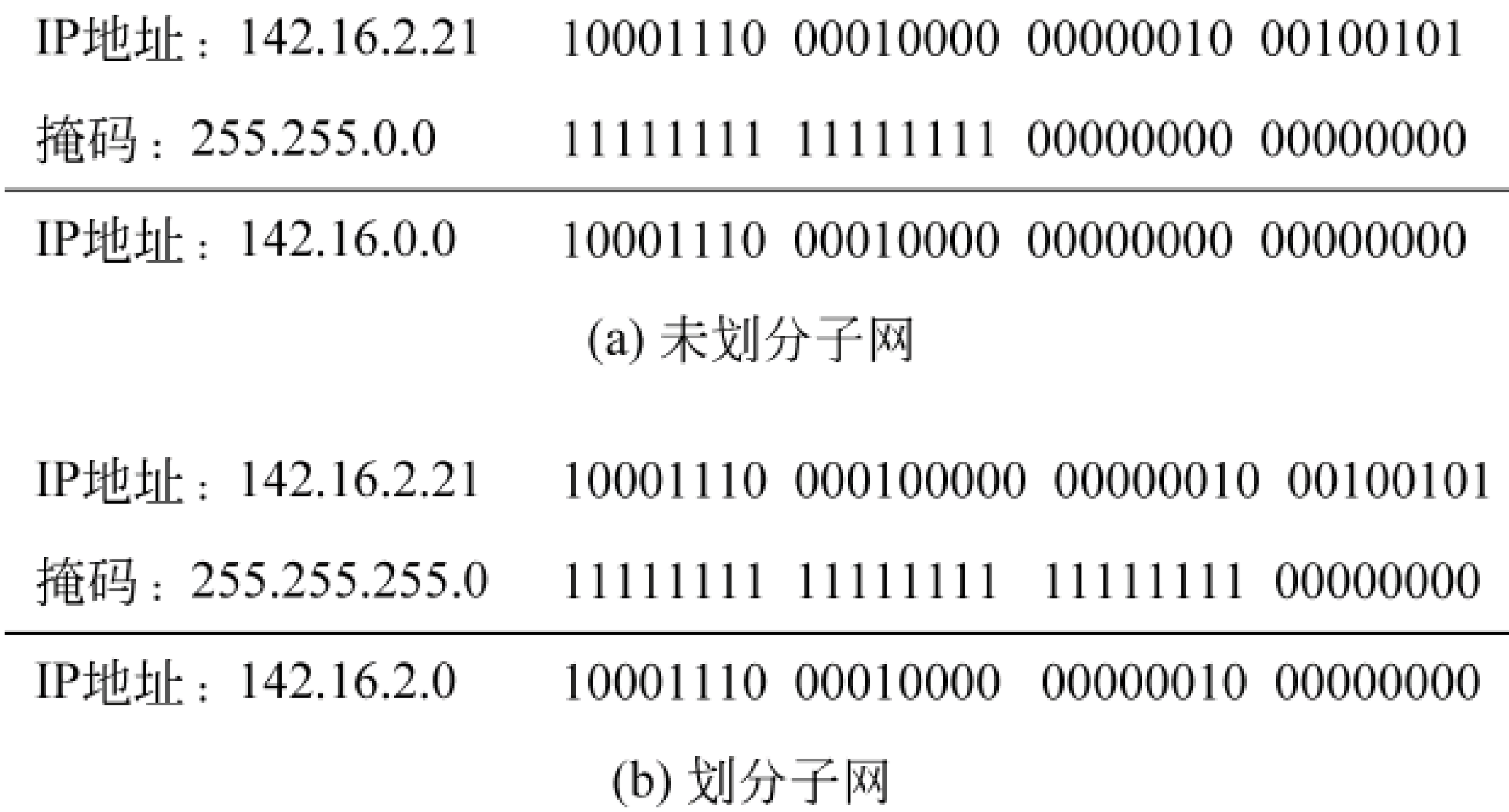


图 5-8　子网掩码的计算过程

2. 无类别域间路由

在可变子网掩码的基础上,人们提出了无类别域间路由(Classless Inter-Domain Routing,CIDR)的概念。RFC1517～RFC1520 文档定义了 CIDR 技术,并且已经形成 Internet 建议标准。CIDR 将剩余的 IP 地址按照可变大小的地址块来分配。与传统的标准分类 IP 地址与子网地址划分的方式相比,CIDR 是以任意的二进制倍数的大小来分配地址的。

由于 CIDR 不采用传统的标准 IP 地址分类方法,因此无法从地址本身来判定网络号的长度。CIDR 地址采用“斜线记法”,即＜IP 地址＞/＜网络前缀＞。例如,用 CIDR 方法给

出一个 IP 地址是 200.16.23.1/20,则表示该 IP 地址的前 20 位是网络前缀,后 12 位是主机号,这个 CIDR 地址结构为 11001000000100000001011100000001。与标准分类 IP 地址一样,主机号为全 0 的网络地址,以及主机号为全 1 的广播地址不分配给主机。

CIDR 将网络前缀相同的、连续的 IP 地址组成一个“CIDR 地址块”。200.16.23.1/20 的网络前缀为 20 位,则该地址块包含的主机数可达 $2^{12}=4096$ 个。一个 CIDR 地址块由块起始地址与前缀来表示。地址块的起始地址是指其中地址数值最小(主机号为全 0)的一个。例如,200.16.23.1/20 地址块中起始地址的结构为 200.16.16.0/20 = 110010000001000000010000 00000000。这个地址块中最大地址的主机号为全 1,其结构为 200.16.31.255/20=11001000 000100000001111111111111。

3. 网络地址转换

由于从 IPv4 过渡到 IPv6 的进程很缓慢,因此需要一种短时间内有效缓解 IP 地址短缺的办法,那就是网络地址转换(Network Address Translation,NAT)技术。NAT 主要用于 4 类应用的动态 IP 地址分配:ISP、ADSL、有线电视与无线移动接入。在使用专用 IP 地址设计的内部网络中,如果内部网络的主机要访问 Internet 或外部网络服务器,这时也需要使用 NAT 技术。

为 ADSL 用户提供拨号服务的 ISP 使用 NAT 技术可以实现 IP 地址的重用。例如,ISP 有 1000 个全局 IP 地址,但是它有 5000 个使用专用 IP 地址的用户。ISP 在具有 NAT 功能的路由器中保持一个 IP 地址池,管理着多个全局 IP 地址。凡是需要访问外部 Internet 服务器的用户首先向 NAT 路由器申请,由 NAT 以动态方式从 IP 地址池中临时分配一个全局 IP 地址给用户;用户访问结束后,NAT 路由器收回 IP 地址,供其他用户使用。这种方式属于多对多的动态映射方式。

为了正确地实现 NAT 功能,NAT 设备必须维护两个地址空间的对应关系,即内部专用 IP 地址与外部全局 IP 地址在转换过程中的对应关系。在实际应用中存在两种可能方法:一种方法是只完成专用 IP 地址与全局 IP 地址转换,这种方法称为网络地址转换(NAT);另一种方法是在专用 IP 地址与全局 IP 地址转换的同时,转换传输层的 TCP 或 UDP 的端口号,这种方法称为网络地址端口变换(NAPT)。

5.3 例题分析

5.3.1 设计要求

根据 IPv4 协议规定的 IP 地址的标准格式,编写程序对输入的 IP 地址进行分析,判断 IP 地址的合法性与地址类型,但是整个过程不能借助任何 inet 函数。在本练习中为了简便起见,只需判断 IP 地址是 A 类、B 类还是 C 类地址。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 IpAddress.exe,则程序的命令行格式为:

```
IpAddress ip_address
```

其中,ip_address 为输入的 IP 地址。

(2) 要求将 IP 地址的类型显示在控制台上,具体格式为:

```
X.X.X.X 的类型为:A类、B类或 C类
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

5.3.2 关键问题

1. 判断 IP 地址的合法性

首先,需要判断输入的 IP 地址的合法性,整个过程要自行编写函数执行判断过程,而不能使用系统提供 inet 系列函数。在设计判断函数时需要全面考虑,以判断不符合 IP 地址书写格式的各类情况。一般来说,可先检查那些明显的错误。例如,判断 IP 地址的总长度是否超过 15。表 5-2 给出了 IP 地址的格式错误。

表 5-2 IP 地址的格式错误

错误类型	错误例子
IP 地址总长度超过 15 位	123.234.123.2345
IP 地址中有不合法的字符	123$234.123.234
IP 地址中分隔符个数不为 3	123.12.123
IP 地址中分隔符连续出现	123.12..123
IP 地址的最后位置为分隔符	123.12.123.
IP 地址的第一个数字为 0	0.234.123.234
IP 地址的任一数字超过 4 位	1.234.123.2345
IP 地址的任一数字大于 255	1.234.123.256

如果需要判断 IP 地址的每段数字是否出错,首先以英文句点“.”为标志将 IP 地址字符串分段,然后将每段字符串转换为对应的整数,最后判断每个数字是否为 0～255 间的整数。这里的关键是如何将字符串按“.”分段。

下面给出 IP 地址分段与判断的伪代码:

```
char temp[4][15];
int ip[4];
//获得 IP 地址每位内容
for(int k=0;k<strlen(ipaddr);k++)
{
    if(ipaddr[k]!='.')
    {
        temp[i][j]=ipaddr[k];
        j++;
    }
```

```
        else
        {
            i++;
            j=0;
        }
    }
    for(i=0;i<4;i++)
    {
        len=strlen(temp[i]);
        //检查每段长度是否大于 3
        if(len>3)
            ...
        //将 IP 地址每段转换为数字
        switch(len)
            ...
        //检查每个数字是否大于 255
        for(i=0;i<4;i++)
            if(ip[i]>255)
                ...
    }
```

2. 判断 IP 地址的类型

这时，需要判断输入的 IP 地址的具体类型，这个判断是在 IP 地址格式合法的基础上。A 类地址的范围为 1.0.0.0～127.255.255.255；B 类地址的范围为 128.0.0.0～191.255.255.255；C 类地址的范围为 192.0.0.0～223.255.255.255。对于那些格式合法但不在此三个范围内的 IP 地址，它们将被标记为其他类型的 IP 地址。

下面给出判断 IP 地址类型的伪代码：

```
//IP 地址为 A 类地址
if(ip[0]>=1&&ip[0]<=127)
    ...
//IP 地址为 B 类地址
if(ip[0]>=128&&ip[0]<=191)
    ...
//IP 地址为 C 类地址
if(ip[0]>=192&&ip[0]<=223)
    ...
//IP 地址为其他类型
if(ip[0]>=224)
    ...
```

3. 程序流程图

图 5-9 给出了主程序流程图。要求输入的命令行参数必须正确，除了程序本身的名称以外，还需要有一个输入的 IP 地址。如果命令行参数的个数不是一个，则程序在输出错误

信息后退出。在主程序的流程中,还需要判断以下几种格式问题:总长度是否超过15、是否有非法字符、分隔符是否为3个、分隔符是否连续、分隔符是否位于结尾、每段长度是否大于3,以及每个数字是否大于255。

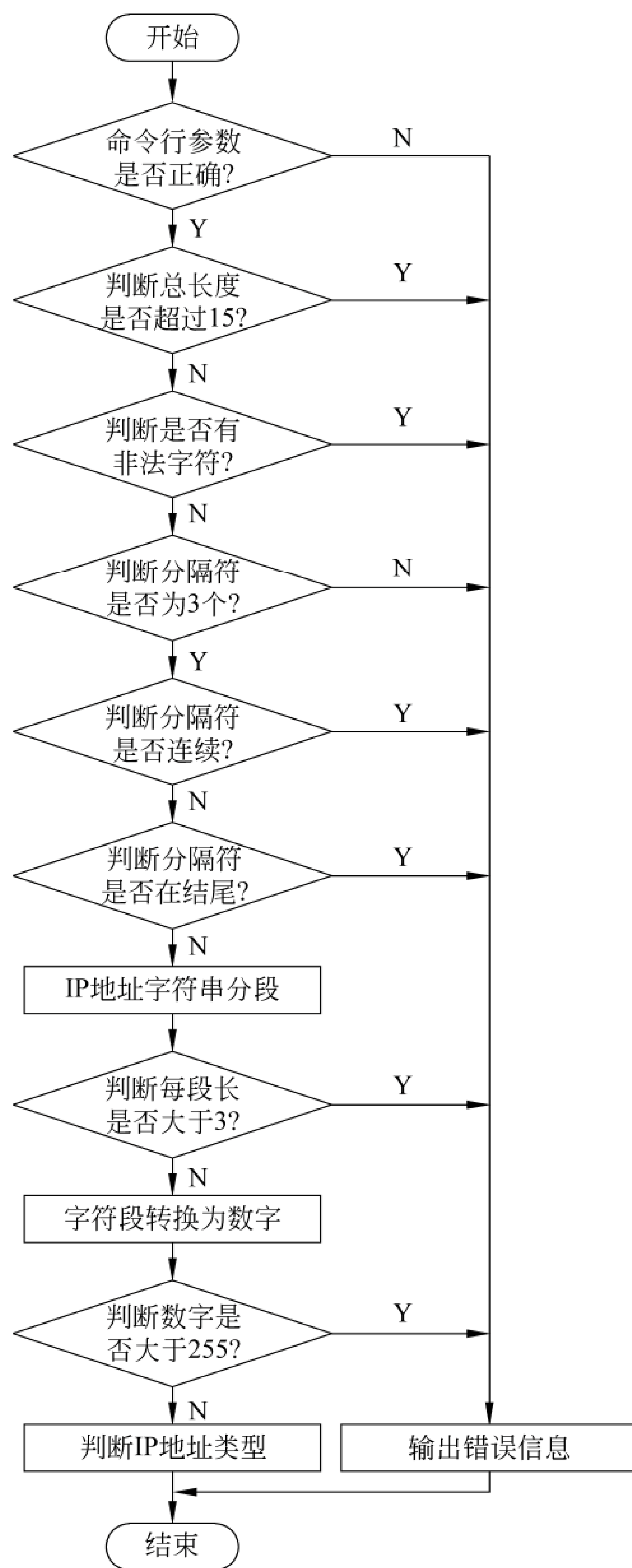


图 5-9　主程序流程图

5.3.3　程序源代码

下面给出IP地址判断程序的源代码:

```
//IpAddress.cpp：定义控制台应用程序的入口点

#include "stdafx.h"
#include "ctype.h"
#include "math.h"
#include "string.h"
#include "iostream"
using namespace std;

void main(int argc, char* argv[])
{
    if(argc!=2)                                    //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:IpAddress ip_address"<<endl;
        return;
    }

    char* ipaddr=new char[strlen(argv[1])];
    strcpy(ipaddr,argv[1]);                        //从命令行复制 IP 地址

    cout<<endl<<"开始分析 IP 地址";
    if(strlen(ipaddr)>15)                          //检查 IP 地址长度
    {
        cout<<endl<<"IP 地址总长度不能超过 15!"<<endl;
        return;
    }

    int dotnum=0;
    for(int i=0;i<strlen(ipaddr);i++)              //检查 IP 地址每位合法性
    {
        if(isdigit(ipaddr[i])==0&&ipaddr[i]!='.')
        {
            cout<<endl<<"IP 地址中含有非法字符!"<<endl;
            return;
        }
        if(ipaddr[i]=='.')                         //IP 地址分隔符数累加
            dotnum++;
    }

    if(dotnum!=3)                                  //检查 IP 地址分隔符数
    {
        cout<<endl<<"IP 地址中分隔符只能为 3 个!"<<endl;
        return;
    }
```

```
for(int i=0;i<strlen(ipaddr)-1;i++)          //检查分隔符是否连续
{
    if(ipaddr[i]=='.'&&ipaddr[i+1]=='.')
    {
        cout<<endl<<"IP 地址中出现连续分隔符!"<<endl;
        return;
    }
}

int len;
len=strlen(ipaddr);
if(ipaddr[len-1]=='.')                        //检查 IP 地址最后位
{
    cout<<endl<<"IP 地址最后位不能为分隔符!"<<endl;
    return;
}

char temp[4][15];                             //初始化 IP 地址缓冲区
for(int i=0;i<4;i++)
    for(int j=0;j<15;j++)
        temp[i][j]='\0';

int ip[4]={0,0,0,0};
int j=0;
int i=0;
for(int k=0;k<strlen(ipaddr);k++)             //获得 IP 地址每位内容
{
    if(ipaddr[k]!='.')
    {
        temp[i][j]=ipaddr[k];
        j++;
    }
    else
    {
        i++;
        j=0;
    }
}

for(int i=0;i<4;i++)                          //IP 地址每位转换为数字
{
    len=strlen(temp[i]);
    if(len>3)                                 //检查 IP 地址每位长度
    {
        cout<<endl<<"IP 地址每位长度不能超过 3!"<<endl;
```

```
            return;
        }

        switch(len)                              //将 IP 地址每位转换为数字
        {
            case 3:                              //IP 地址该位为 100~255
            {
                while(len!=0)
                {
                    ip[i]+=(temp[i][len-1]-48)*pow(10,3-len);
                    len--;
                }
                break;
            }
            case 2:                              //IP 地址该位为 10~99
            {
                while(len!=0)
                {
                    ip[i]+=(temp[i][len-1]-48)*pow(10,2-len);
                    len--;
                }
                break;
            }
            default:                             //IP 地址该位为 1~9
            {
                while(len!=0)
                {
                    ip[i]+=(temp[i][len-1]-48)*pow(10,1-len);
                    len--;
                }
            }
        }
    }

    for(int i=0;i<4;i++)
    {
        if(ip[i]>255)                            //检查 IP 地址每位数字
        {
            cout<<endl<<"IP 地址数字不能超过 255!"<<endl;
            return;
        }
    }

    if(ip[0]<1)                                  //IP 地址首位不能为 0
        cout<<endl<<"IP 地址首位不能为 0!"<<endl;
```

```
        if(ip[0]>=1&&ip[0]<=127)                    //IP 地址为 A 类地址
            cout<<endl<<argv[1]<<"是 A 类 IP 地址!"<<endl;
        if(ip[0]>=128&&ip[0]<=191)                  //IP 地址为 B 类地址
            cout<<endl<<argv[1]<<"是 B 类 IP 地址!"<<endl;
        if(ip[0]>=192&&ip[0]<=223)                  //IP 地址为 C 类地址
            cout<<endl<<argv[1]<<"是 C 类 IP 地址!"<<endl;
        if(ip[0]>=224)                              //IP 地址为其他类型
            cout<<endl<<argv[1]<<"是其他类型 IP 地址!"<<endl;

        return;
    }
```

图5-10给出了IP地址的类型判断。程序命令行输入为IpAddress ip_address。程序对输入的IP地址进行合法性判断,然后对合法的IP地址进行类型判断,并将得到的结果显示在控制台上。

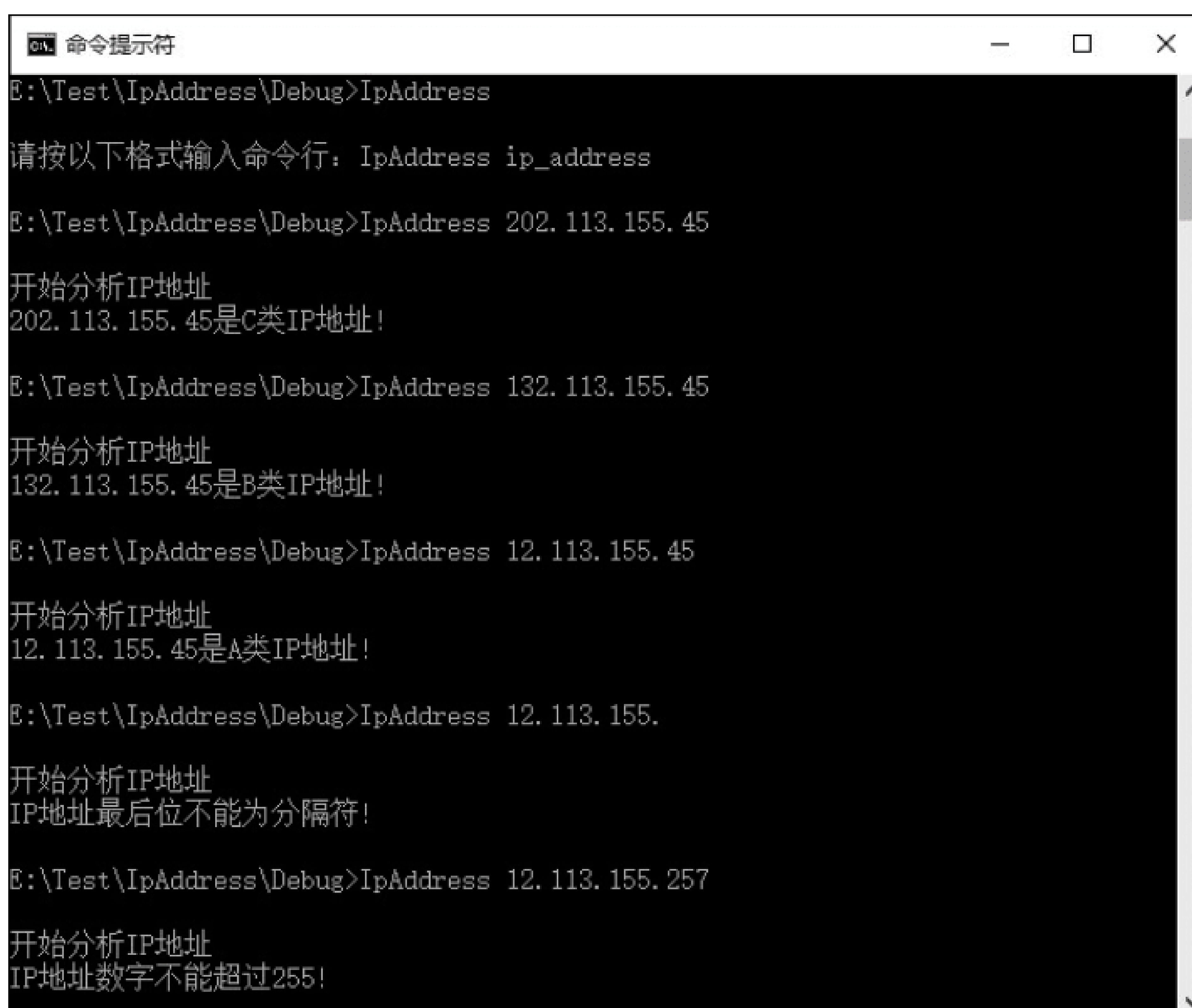


图5-10　IP地址的类型判断

5.4　练　习　题

根据IPv4协议规定的IP地址的标准格式,编写程序对输入的IP地址与掩码进行合法性检查,并将计算得到的子网地址显示在控制台上,但是整个过程不能借助任何inet函数。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为IpAddress.exe,则程序的命令行格式为:

```
IpAddress ip_address mask
```

其中,ip_address 为输入的 IP 地址,mask 为子网掩码。

(2) 要求将计算的子网地址显示在控制台上,具体格式为:

```
子网地址为:X.X.X.X
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 6 章 IP 数据包的捕获与解析

6.1 设计目的

IP 包是在网络层中进行数据传输的基本单位。熟悉 IP 包结构，对于理解网络协议的概念、网络层次结构、协议执行过程以及网络问题处理的一般方法，具有重要的意义。本章练习的目的是根据网络层的基本原理，通过网卡截获与解析标准格式的 IP 包，了解 IP 头部中各个字段的含义与用途，从而深入理解网络协议的工作原理。

6.2 相关知识

本章涉及的相关知识包括网络层的概念与 IP 数据包的结构。

6.2.1 网络层的基本概念

网络中的两台主机之间通信要遵循相同的网络协议。各种异构网络的互联带来了协议的标准化问题，这就促进了网络体系结构与参考模型的研究。OSI 参考模型是由 ISO 组织制定的一个网络互联参考模型。网络层是 OSI 参考模型的一个重要层次，它需要使用下面的数据链路层的服务，并且为上面的传输层提供服务。网络层的主要功能是：通过路由选择算法，为分组通过通信子网选择最适当的路径，实现拥塞控制、流量控制与网络互联等功能。OSI 参考模型的每层都有自己的数据传输单位，在经过不同层时需要组装成符合要求的单位。分组（packet）是网络层传输的基本单位。

TCP/IP 参考模型是一个重要的网络参考模型。在 TCP/IP 协议的研究初期，并没有提出相应的参考模型。随着 TCP/IP 协议获得学术界的广泛认可，研究者在 1974 年定义了最初的参考模型，并在 1988 年完善了 TCP/IP 参考模型。目前，TCP/IP 协议受到计算机产业界的支持，TCP/IP 参考模型已成为事实上的标准。TCP/IP 参考模型从下到上依次是：主机-网络层、互联层、传输层、应用层。其中，互联层对应于 OSI 参考模型的网络层，它提供的功能也与网络层基本一致。图 6-1 给出了 TCP/IP 协议族的结构。TCP/IP 参考模型的各层都有不同的协议，这些协议共同构成了 TCP/IP 协议族。

TCP/IP 协议体系主要有以下特点。

- 开放的协议标准。

应用层	Telnet	FTP	SMTP	DNS	其他协议
传输层	TCP		UDP		
互联层	IP				
	ARP	RARP			
主机-网络层	Ethernet	Token Ring	其他协议		

图 6-1　TCP/IP 协议族的结构

- 独立于特定的计算机硬件与操作系统。
- 独立于特定的网络硬件,可运行在局域网、广域网,更适用于 Internet。
- 统一的网络地址分配方案,所有网络设备在 Internet 中都有唯一的 IP 地址。
- 标准化的应用层协议,可提供多种拥有大量用户的网络服务。

6.2.2　IP 数据包的结构

IP 协议是 TCP/IP 协议体系中的核心协议。IP 协议制定了统一的 IP 数据包格式,以消除 Internet 中各种通信子网之间的差异,为数据的收发双方提供透明的传输通道。RFC791 是最早出现的 IP 协议文档,它描述了 IPv4 协议的基本内容,主要是 IPv4 数据包结构。这个 RFC 文档一直沿用至今,后期仅出现一些补充性质的文档。IPv4 数据包又称为 IPv4 分组或 IPv4 数据报,它们在概念上是相同的。IPv4 数据包的长度是可变的,它分为头部与数据两个部分。图 6-2 给出了 IPv4 数据包的结构。IPv4 头部的长度为 20～60B,通常以 4B 为单位来表示头部字段。



图 6-2　IPv4 数据包的结构

IP 数据包头部由以下字段组成。

1. 版本

版本(version)字段的长度为 4 位,表示使用的 IP 协议的版本。目前协议的版本是 IPv4,版本字段值为 4;下一代协议的版本是 IPv6,版本字段值为 6。该字段向网络层软件说明 IP 数据包的版本,不同的版本的数据包的结构不同。IP 软件在处理数据包之前必须检查版本号。本程序只是针对 IPv4 数据包的解析。

2. 头部长度

头部长度(header length)字段的长度为4位,表示IPv4头部的长度。IPv4头部中除了选项和填充字段之外,其他字段的长度都是固定的。IPv4头部的基本长度为20B,如果加上最长40B的选项字段,则IPv4头部的最大长度为60B。

3. 服务类型

服务类型(service type)字段的长度为8位,表示路由器如何处理这个IPv4数据包。图6-3给出了服务类型字段的结构。服务类型字段长度由三部分构成:3位优先级、4位服务类型与1位保留位。

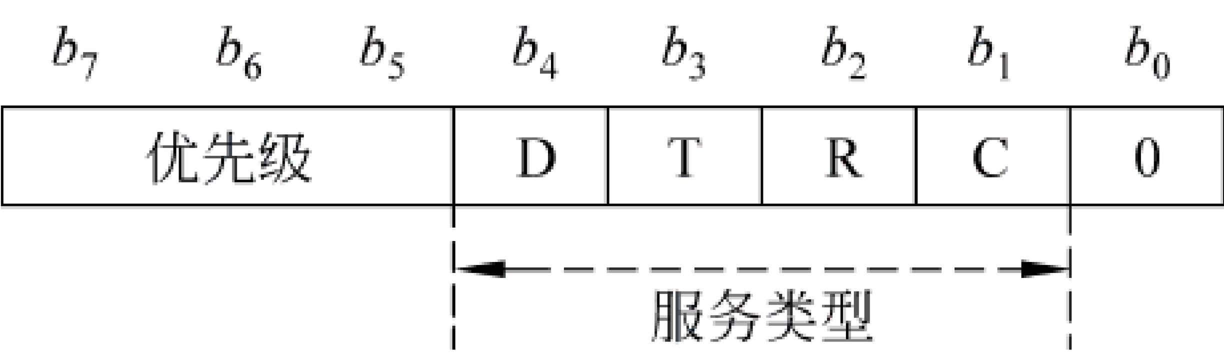


图6-3 服务类型字段的结构

(1) 优先级

优先级(precedence)为该字节的最高3位,表示这个IPv4数据包的重要性。优先级共分为8个等级,优先级越高的数据包越重要,则路由器需要对它进行优先处理。当路由器处于严重的拥塞状态时,它会接收优先级高的数据包,而丢弃优先级低的数据包。表6-1给出了优先级部分的说明。

表6-1 优先级部分的说明

位数($b_7b_6b_5$)	意 义
111	网络控制(network control)
110	互联网络控制(internetwork control)
101	重要(critic)
100	即时优先(flash override)
011	即时(flash)
010	立刻(immediate)
001	优先(priority)
000	普通(routine)

(2) 服务类型

服务类型(type of service)为该字节紧接着优先级的4位,与优先级共同表示IPv4数据包的重要性。这里,b_4、b_3、b_2、b_1分别表示:D(延迟)、T(吞吐量)、R(可靠性)与C(成本)。每个组合的服务类型的4位中,最多只能有一位的值为1,其他三位的值为0。表6-2给出了服务类型部分的说明。

表6-2 服务类型部分的说明

位数($b_4b_3b_2b_1$)	意 义
1111	安全级最高(maximize security)
1000	延迟最小(minimize delay)
0100	吞吐量最大(maximize throughput)

续表

位数($b_4b_3b_2b_1$)	意　　义
0010	可靠性最大(maximize reliability)
0001	成本最小(minimize cost)
0000	普通服务(normal service)

4. 总长度

总长度(total length)字段的长度为16位,表示以字节为单位的IPv4数据包总长度。由于这个字段的长度为16位,因此IPv4数据包的最大长度为$(2^{16}-1)$B=65 535B。数据部分长度为总长度减去头部长度。

5. 标识符

标识符(identification)字段长度为16位,表示这个分片属于哪个IPv4数据包。IPv4数据包的所有分片可分配一个标识符,最多可分配的值为65 535个。标识符、标志位与片偏移等字段共同用于IP数据包的分片。

6. 标志位

标志位(flags)字段的长度为16位,表示IPv4数据包是否可以分片。图7-2给出了标志位字段的结构。标志位字段由三部分构成:保留位、DF位与MF位。其中,最高位为保留位,取值为0;中间位是不分片位(Do not Fragment,DF),1表示不能分片,0表示可以分片;最低位是更多分片位(More Fragment,MF),1表示不是最后一个分片,0表示最后一个分片。标识符、标志位与片偏移等字段共同用于IP数据包的分片。

7. 片偏移

片偏移(fragment offset)字段长度为13位,表示分片在整个IPv4包中的相对位置。片偏移值是以8B为基本单位来计数,因此分片长度应该是8B的整数倍。标识符、标志位与片偏移等字段共同用于IP数据包的分片。

8. 生存周期

生存周期(time to live)字段的长度为8位,表示IPv4数据包在传输过程中的寿命,通常用经过的路由器最大跳步数来限制。IP数据包从源主机到目的主机的传输延时不确定,为了避免由于出错而造成IP数据包的循环传输,可以通过设置生存周期来解决这个问题。TTL的初始值由源主机设置,经过一个路由器转发之后,TTL值减1。当TTL的值为0时,丢弃分组并发送ICMP报文通知源主机。

9. 协议

协议(protocol)字段的长度为8位,表示IPv4数据包的高层协议类型。表6-3给出了常用的高层协议。通过分析协议字段的具体数值,可看出数据部分封装的高层协议,可能包括传输层协议(TCP、UDP等)或网络层的辅助协议(ICMP、IGMP等)。例如,协议字段的数值为6,表示数据部分封装的是TCP报文段。

10. 头部校验和

头部校验和(header checksum)字段的长度为16位,用来检查IPv4数据包在传输中是否出错。头部校验和的计算方法为:将头部校验和字段的值置为0,将IPv4头部的值以16位为单位进行累加(异或),再将累加结果取补码写入头部校验和字段。

表 6-3 常用的高层协议

字 段 值	协 议 名 称	字 段 值	协 议 名 称
1	ICMP	17	UDP
2	IGMP	41	IPv6
6	TCP	46	RSVP
8	EGP	89	OSPF

11. IP 地址

IP 地址(IP address)包括两个部分：源 IP 地址与目的 IP 地址。这里，源 IP 地址与目的 IP 地址的长度均为 32 位。源 IP 地址是发送数据包的源主机 IPv4 地址；目的 IP 地址是接收数据包的目的主机 IPv4 地址。

12. 选项

选项(options)字段的长度范围是 0～40B，主要用于控制和测试两个目的。选项由三部分构成：选项码、长度与选项数据。这里，选项码用于确定该选项的具体功能，例如源路由、记录路由、时间戳等；长度表示选项数据的大小；选项数据的具体内容由不同功能来决定。在使用选项字段的过程中，可能出现头部长度不是 32 位整数倍的情况，这时就需要通过填充(0)来凑齐相应的位数。

6.3 例题分析

6.3.1 设计要求

编写程序来捕获网络中传输的 IPv4 数据包，并将得到的解析结果显示出来。在本练习中为了简便起见，只解析 IPv4 头部中除选项外的各字段值，不需要解析出具体的服务类型与协议类型。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 PackParse.exe，则程序的命令行格式为：

```
PackParse packet_sum
```

其中，packet_sum 为捕获 IPv4 数据包的数量。

(2) 要求将部分字段内容显示在控制台上，具体格式为：

```
------------------------------------
版本:xx
头部长度:xx
服务类型:xx,xx
总长度:xx
标识符:xx
标志位:xx,DF,MF
片偏移:xx
生存周期:xx
```

```
协议:xx
头部校验和:xx
源 IP 地址:xx.xx.xx.xx
目的 IP 地址:xx.xx.xx.xx
------------------------------------
…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

6.3.2 关键问题

1. 创建原始套接字

为了通过网卡来截获传输中的 IPv4 数据包,这时需要对套接字(socket)进行编程。套接字可以分为三种类型:流套接字(stream socket)、数据包套接字(datagram socket)和原始套接字(raw socket)。其中,流套接字与数据包套接字都是用于网络通信,它们只能响应与本地网卡的硬件地址匹配或广播的数据包;而原始套接字就不受这些限制,可接收与本地网卡的硬件地址不匹配的数据包,在本练习中需要使用这种套接字。

下面给出创建原始套接字的伪代码:

```
//套接字异步启动
WSADATA WSAData;
if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)
    …
//创建原始 Socket
SOCKET sock;
if((sock=socket(AF_INET,SOCK_RAW,IPPROTO_IP))==INVALID_SOCKET)
    …
```

socket 函数的第一个参数指定使用的协议族,对于 TCP/IP 协议族,该参数设置为 AF_INET;第二个参数指定创建的套接字类型,SOCK_STREAM、SOCK_DGRAM 与 SOCK_RAW 分别为流式套接字、数据报套接字与原始套接字,这里设置为 SOCK_RAW;第三个参数指定使用的通信协议,该参数需要依赖于第二个参数,这里设置为 IPPROTO_IP。如果该函数执行成功,返回新创建的套接字描述符;否则,返回 INVALID_SOCKET。

2. 初始化 Socket 结构

在成功地创建原始套接字以后,首先要设置对 IPv4 头部的操作模式,这时就需要调用 setsockopt 函数。该函数的第一个参数指定使用的套接字,这里使用刚创建的套接字描述符;第二个参数指定使用的通信协议,这里设置为 IPPROTO_IP;后三个参数指定如何对数据包进行操作,第三个参数设置为 IP_HDRINCL,并将 flag 设置为 true,表示用户自己处理 IPv4 头部。

下面给出设置 IP 头部操作选项的伪代码:

```
BOOL flag=true;
setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char * )&flag,sizeof(flag));
```

在进行原始套接字的初始化时,sockaddr_in 的地址值应为本机的 IP 地址,它可以通过 gethostbyname 函数来获取;端口号可随便填写;协议族应填为 AF_INET。需要注意的是,填写 sockaddr_in 结构的值必须是以网络字节顺序表示的值,而不能直接使用本机字节顺序的值,用 htons 函数可将主机数据转换为网络字节顺序的数据。最后,需要调用 bind 函数将 Socket 绑定到本地网卡。

下面给出对 Socket 进行初始化的伪代码:

```
//获得本地主机名
char hostName[128];
gethostname(hostName,100);
//获得本地 IP 地址
hostent * pHostIP;
pHostIP=gethostbyname(hostName);
//填充 sockaddr_in
sockaddr_in addr_in;
addr_in.sin_family=AF_INET;
addr_in.sin_port=htons(6000);
addr_in.sin_addr= * (in_addr * )pHostIP->h_addr_list[0];
//Socket 绑定本地网卡
bind(sock,(PSOCKADDR)&addr_in,sizeof(addr_in));
```

3. 接收 IPv4 数据包

在调用相关函数接收 IPv4 数据包之前,需要注意的是,通常网卡不能接收目的地址不是自己的数据包。也就是说,应用程序无法接收与自己无关的数据包。如果想截获流经网卡的所有 IPv4 数据包,需要调用 WSAIoctl 函数将网卡设置为混杂模式。这时,当接收的数据包中的协议类型与原始套接字匹配,接收的数据包就被复制到套接字的缓冲区中。

下面给出将网卡设置为混杂模式的伪代码:

```
#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
DWORD dwBufferLen[10];
DWORD dwBufferInLen=1;
DWORD dwBytesReturned=0;
WSAIoctl(SnifferSocket,IO_RCVALL,&dwBufferInLen,sizeof(dwBufferInLen),
&dwBufferLen,sizeof(dwBufferLen),&dwBytesReturned,NULL,NULL);
```

接着,需要调用 recv 函数来接收 IPv4 数据包。该函数的第一个参数指定使用的套接字描述符;第二个参数是接收缓冲区的地址;第三个参数是接收缓冲区的大小,也就是要接收的字节数;第四个参数是一个附加标志,如果对接收数据没有特殊要求,可设置为 0。由于 IPv4 数据包的最大长度是 65 535B,因此缓冲区的大小不能小于 65 535。在设置合适的

接收缓冲区后，可利用循环来反复监听与捕获 IP 数据包。

下面给出捕获 IP 数据包的代码：

```
//设置缓冲区
#define BUFFER_SIZE 65535
char buffer[BUFFER_SIZE];
//接收 IP 数据包
while(数据包捕获未结束)
{
    recv(sock,buffer,BUFFER_SIZE,0);
    …
}
```

4. 定义 IPv4 头部数据结构

在对 IPv4 头部各字段进行解析之前，首先需要构造一个 IPv4 头部数据结构。这个数据结构要与图 6-2 中的 IPv4 数据包头部结构一致，从字段顺序到每个字段的大小都相同。

下面给出构造 IPv4 头部数据结构的伪代码：

```
//定义 IP 头部结构
typedef struct IP_HEAD
{
    union
    {
        unsigned char Version;
        unsigned char HeadLen;
    };
    unsigned char ServiceType;
    unsigned short TotalLen;
    unsigned short Identifier;
    union
    {
        unsigned short Flags;
        unsigned short FragOffset;
    };
    unsigned char TimeToLive;
    unsigned char Protocol;
    unsigned short HeadChecksum;
    unsigned int SourceAddr;
    unsigned int DestinAddr;
    unsigned char Options;
}ip_head;
```

5. 解析 IPv4 头部字段

由于本练习只解析 IPv4 头部的各字段，并不需要处理 IP 数据包的数据部分，因此可通过指针将缓冲区中的内容强制转化为 ip_head 结构，然后依次取出该结构中的各字段内容。

根据IPv4头部各字段的长度不同，对于那些是8位整数倍(8位、16位与32位)的字段，可以通过ip_head的成员变量直接获得；对于那些不是8位整数倍的字段(如版本)，则需要通过C语言中的移位以及与、或操作来获得。

下面给出解析IPv4头部字段的伪代码：

```
//缓冲区内容强制转换
ip_head ip= * (ip_head * )buffer;
//依次取出各字段内容
ip.Version>>4;
ip.HeadLen&0x0f;
ip.ServiceType>>5;
(ip.ServiceType>>1)&0x0f;
ip.TotalLen;
ip.Identifier;
(ip.Flags>>14)&0x01;
(ip.Flags>>13)&0x01;
ip.FragOffset&0x1fff;
ip.TimeToLive;
ip.Protocol;
ip.HeadChecksum;
inet_ntoa(* (in_addr * )&ip.SourceAddr);
inet_ntoa(* (in_addr * )&ip.DestAddr);
```

6. 程序流程图

图6-4给出了主程序流程图。要求输入的命令行参数必须正确，除了程序本身的名称以外，还需要有一个捕获数据包的数量作为参数。如果命令行参数的个数不是一个，则程序在输出错误信息后退出。在主程序流程中，需要判断是否捕获IPv4数据包。

6.3.3 程序源代码

下面给出IP数据包捕获程序的源代码：

```
//PackParse.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "winsock2.h"
#include "ws2tcpip.h"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib
#define IO_RCVALL _WSAIOW(IOC_VENDOR,1)

typedef struct IP_HEAD                           //定义 IP 包头部结构
{
```

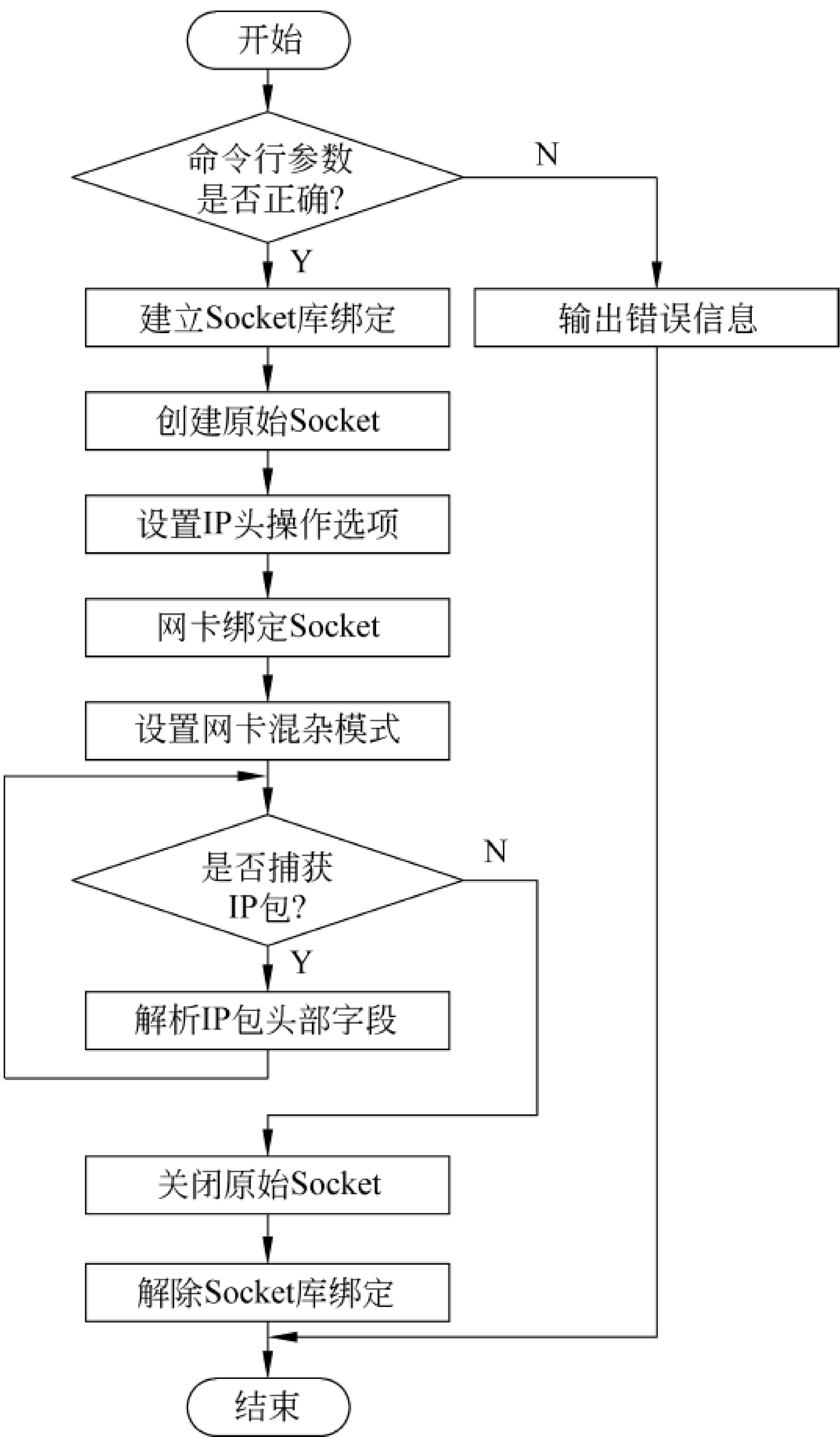


图 6-4　主程序流程图

```
union
{
    unsigned char Version;                    //版本(字节前4位)
    unsigned char HeadLen;                    //头部长度(字节后4位)
};
unsigned char ServiceType;                    //服务类型
unsigned short TotalLen;                      //总长度
unsigned short Identifier;                    //标识符
union
{
    unsigned short Flags;                     //标志位(字前3位)
    unsigned short FragOffset;                //片偏移(字后13位)
};
unsigned char TimeToLive;                     //生存周期
unsigned char Protocol;                       //协议
unsigned short HeadChecksum;                  //头部校验和
```

```
    unsigned int SourceAddr;                          //源 IP 地址
    unsigned int DestinAddr;                          //目的 IP 地址
    unsigned char Options;                            //选项
}ip_head;

void main(int argc, char * argv[])
{
    if(argc!=2)                                       //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:PackParse packet_sum"<<endl;
        return;
    }

    WSADATA WSAData;
    if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)         //套接字异步启动
    {
        cout<<endl<<"WSAStartup 初始化失败"<<endl;
        return;
    }

    SOCKET sock=socket(AF_INET,SOCK_RAW,IPPROTO_IP);
    if(sock==INVALID_SOCKET)                          //创建原始 Socket
    {
        cout<<endl<<"创建 Socket 失败!"<<endl;
        return;
    }

    BOOL flag=true;                                   //设置 IP 头操作选项
    if(setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char * )&flag,sizeof(flag))==
    SOCKET_ERROR)
    {
        cout<<endl<<"setsockopt 操作失败"<<endl;
        return;
    }

    char hostName[128];
    if(gethostname(hostName,100)==SOCKET_ERROR)       //获得本地主机名
    {
        cout<<endl<<"gethostname 操作失败"<<endl;
        return;
    }

    hostent * pHostIP;
    if((pHostIP=gethostbyname(hostName))==NULL)       //获取本地 IP 地址
```

```
{
    cout<<endl<<"gethostbyname 操作失败"<<endl;
return;
}

sockaddr_in host_addr;                          //填充 sockaddr_in
host_addr.sin_addr= * (in_addr * )pHostIP->h_addr_list[0];
host_addr.sin_family=AF_INET;
host_addr.sin_port=htons(6000);

if(bind(sock,(PSOCKADDR)&host_addr,sizeof(host_addr))==SOCKET_ERROR)
{                                               //Socket 绑定本地网卡
    cout<<endl<<"bind 操作失败"<<endl;
    return;
}

DWORD dwBufferLen[10];
DWORD dwBufferInLen=1;
DWORD dwBytesReturned=0;
if(WSAIoctl(sock,IO_RCVALL,&dwBufferInLen,sizeof(dwBufferInLen),
&dwBufferLen,sizeof(dwBufferLen),&dwBytesReturned,NULL,NULL)==SOCKET_
ERROR)
{                                               //设置 Socket 接收所有包
    cout<<endl<<"WSAIoctl 操作失败"<<endl;
    return;
}

cout<<endl<<"开始解析 IP 包:"<<endl;
char buffer[65535];                             //设置缓冲区大小
int packsum=atoi(argv[1]);
for(int i=0;i<packsum;i++)                      //开始接收 IP 包
{
    if(recv(sock,buffer,65535,0)>0)
    {
        ip_head ip= * (ip_head * )buffer;       //逐位解析 IP 头部字段
        cout<<"-----------------------------------------"<<endl;
        cout<<"版本:"<<(ip.Version>>4)<<endl;
        cout<<"头部长度:"<<((ip.HeadLen&0x0f) * 4)<<endl;
        cout<<"服务类型:Priority "<<(ip.ServiceType>>5)<<",Service "<<((ip.
        ServiceType>>1)&0x0f)<<endl;
        cout<<"总长度:"<<ip.TotalLen<<endl;
        cout<<"标识符:"<<ip.Identifier<<endl;
        cout<<"标志位:"<<((ip.Flags>>15)&0x01)<<",DF="<<((ip.Flags>>14)&0x01)<
       <",MF="<<((ip.Flags>>13)&0x01)<<"endl;
```

```
                cout<<"片偏移:"<<(ip.FragOffset&0x1fff)<<endl;
                cout<<"生存周期:"<<(int)ip.TimeToLive<<endl;
                cout<<"协议:Protocol "<<(int)ip.Protocol<<endl;
                cout<<"头部校验和:"<<ip.HeadChecksum<<endl;
                cout<<"源 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.SourceAddr)<<endl;
                cout<<"目的 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.DestinAddr)<<endl;
            }
        }

        closesocket(sock);                          //关闭原始 Socket
        WSACleanup();                               //套接字异步关闭
    }
```

图 6-5 给出了 IP 数据包捕获的过程。程序命令行输入为 PackParse 2。程序指定本地网卡以混杂模式捕获两个 IP 数据包,依次解析每个 IP 数据包头部的各个字段,然后将获得的结果显示在控制台上。

```
管理员: 命令提示符
E:\Test\PackParse\Debug>PackParse

请按以下格式输入命令行: PackParse packet_sum

E:\Test\PackParse\Debug>PackParse 2

开始解析IP包:
----------------------------------------
版本: 4
头部长度: 20
服务类型: Priority 0, Service 0
总长度: 33536
标识符: 3383
标志位: 0, DF=0, MF=0
片偏移: 0
生存周期: 64
协议: Protocol 17
头部校验和: 0
源IP地址: 10.134.37.128
目的IP地址: 150.175.60.105
----------------------------------------
版本: 4
头部长度: 20
服务类型: Priority 0, Service 0
总长度: 13312
标识符: 1864
标志位: 0, DF=0, MF=0
片偏移: 64
生存周期: 64
协议: Protocol 6
头部校验和: 0
源IP地址: 10.134.37.128
目的IP地址: 67.56.85.103
```

图 6-5 IP 数据包捕获的过程

6.4 练 习 题

编写程序来捕获网络中传输的 IPv4 数据包,并将获得的解析结果写入输出文件。要求程序在接收到键盘输入 Ctrl+C 时,停止捕获 IP 数据包并退出程序。在本练习中为了简便

起见，只解析 IPv4 头部中除选项外的各字段值，要求解析出具体的服务类型与协议类型。程序设计的具体要求如下。

（1）要求程序为命令行程序。例如，可执行文件名为 PackParse.exe，则程序的命令行格式为：

```
PackParse output_file
```

其中，output_file 为输出文件。

（2）要求将部分字段内容写入输出文件，具体格式为：

```
------------------------------------
版本:xx
头部长度:xx
服务类型:xx,xx
总长度:xx
标识符:xx
标志位:xx,DF,MF
片偏移:xx
生存周期:xx
协议:xx
头部校验和:xx
源 IP 地址:xx.xx.xx.xx
目的 IP 地址:xx.xx.xx.xx
------------------------------------
…
```

（3）要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

（4）要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 7 章 IP数据包的分片与重组

7.1 设计目的

IP 数据包(简称 IP 包)是在网络层进行数据传输的基本单位,但是它在传输过程中经常需要分片与重组。熟悉 IP 数据包的分片与重组,对于理解网络层次结构以及网络问题处理方法,具有重要的意义。本章练习的目的是,根据网络层基本原理,通过对已有 IP 数据包进行分片与重组,了解 IP 数据包结构中各字段的含义与用途,从而深入理解网络层与下面各层的关系。

7.2 相关知识

本章涉及的相关知识包括 IP 数据包的分片的概念与相关字段。

7.2.1 IP 包分片的概念

IP 数据包作为网络层数据必然通过数据链路层,封装成帧再通过物理层来传输。一个 IP 包可能经过多个不同的物理网络,每个路由器需要对接收的帧进行拆帧与处理,然后封装成某种类型的帧来通过物理网络。这个帧的格式与长度取决于物理网络采用的协议。例如,如果路由器接收到一个 Ethernet 帧,而要将它转发到一个 Token Ring 中,则路由器需要按照 Token Ring 协议去构造相应的帧,这两种帧的格式与长度都不相同。每种物理网络都规定了各自帧的数据字段最大长度,称为最大传输单元(Maximum Transfer Unit,MTU)。表 7-1 给出了几种物理网络的最大传输单元。

表 7-1 几种物理网络的最大传输单元

物理网络	最大传输单元(MTU)	说明文档
Token Ring	17 914B	RFC1042
FDDI	4352B	RFC1188
Ethernet	1500B	RFC894
X.25	576B	RFC877
PPP	296B	RFC1144

为了使 IP 协议与物理网络无关,RFC791 文档中规定 IP 包的 MTU 为 65 535B。从传

输层与网络层角度来看,传输层数据加上 IP 头部的总长度必须小于 65 535B,否则需要将传输层数据分别封装在不同 IP 包中。由于多个 IP 包传输出错概率增加,因此在应用层与传输层开始控制数据长度。从网络层与数据链路层角度来看,IP 包的最大长度为 65 535B,则使用的物理网络 MTU 为 65 535B 时效率最高。但是,实际上多数物理网络 MTU 比 IP 包的最大长度短。例如,Ethernet 的 MTU 为 1500B,它远小于 IP 包规定的最大长度。

当使用这些物理网络来传输 IP 包时,需要将 IP 包分成若干个较小的片(fragment)。如果 IP 包来自一个 MTU 较大的网络,当它要通过一个 MTU 较小的网络时,首先必须对 IP 包进行分片处理。图 7-1 给出了 IP 数据包分片的方法。对 IP 数据包分片,首先需要确定分片长度,然后将原始 IP 数据包头部与部分数据构成第 1 个片;如果剩下的数据部分仍超过分片长度,则需要将分片数据加上原来的 IP 数据包头部构成第 2 个片;这样一直分割下去,直到剩下的数据小于分片长度为止。当然,目的节点需要将分片的 IP 包重组成原始 IP 包。

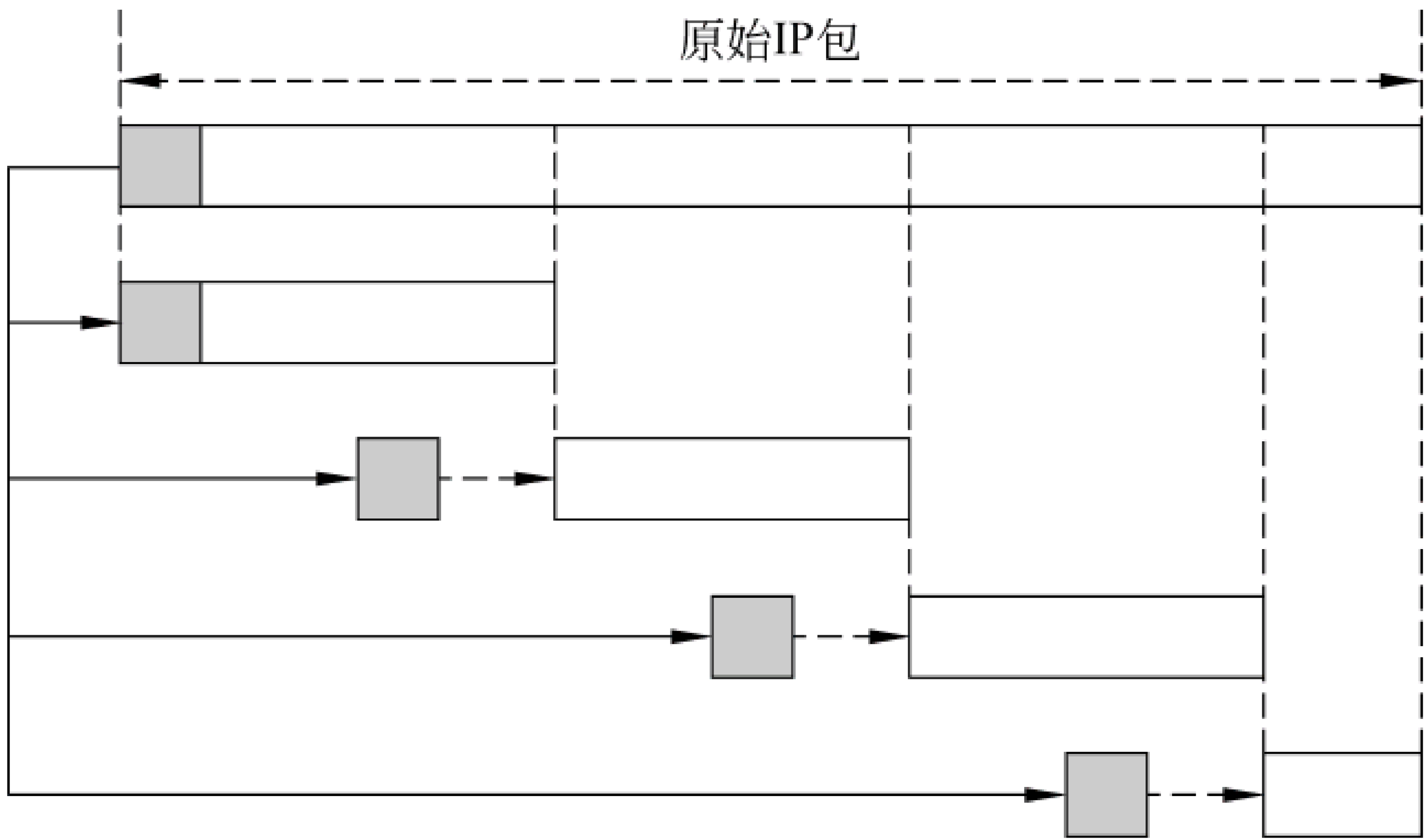


图 7-1　IP 数据包分片的方法

7.2.2　IP 包分片的相关字段

IP 头部与分片、组装相关的字段包括以下内容。

1. 标识符

标识符(identification)字段长度为 16 位,表示 IP 包分片属于哪个 IP 包。一个 IP 包的所有分片可分配一个标识符,最多可以分配的值为 65 535 个。由于 IP 包可能通过不同传输路径到达目的节点,属于同一 IP 包的不同分片到达时可能乱序,或者与属于其他 IP 包的分片混合在一起传输。例如,IP 包的某个分片分配的标识符为 1562,目的节点可以从接收到的各种 IP 包分片中,将所有标识符为 1562 的分片挑选出来重组。

2. 标志位

标志位(flags)字段的长度为 16 位,表示 IP 包是否可以分片。图 7-2 给出了标志位字段的结构。标志位字段由三部分构成:保留位、DF 位与 MF 位。其中,最高位为保留位,取值为 0;中间位是不分片位(Do not Fragment, DF),1 表示不能对 IP 包进行分片,0 表示可以分片;最低



图 7-2　标志位字段的结构

位是更多分片位(More Fragment,MF),1 表示接收的不是最后一个分片,0 表示是最后一个分片。如果 IP 包的长度超过 MTU 大小,同时 IP 包又不能分片,则这个 IP 包只能被丢弃,并发送 ICMP 包向源主机报告。

3. 片偏移

片偏移(fragment offset)字段长度为 13 位,表示分片在整个 IP 包中的相对位置。片偏移值是以 8B 为基本单位来计数,因此分片长度应该是 8B 的整数倍。图 7-3 给出了 IP 包分片的相关字段值。如果原始 IP 包的总长度为 2220B,按 MTU 长度为 820B 可分为三个分片,则分片 1 与分片 2 的长度均为 820B,分片 3 的长度为 620B。分片 1、分片 2 与分片 3 的片偏移分别为 0、100 与 200。在对 IP 包的所有分片进行重组时,需要根据片偏移的值来决定分片的顺序。

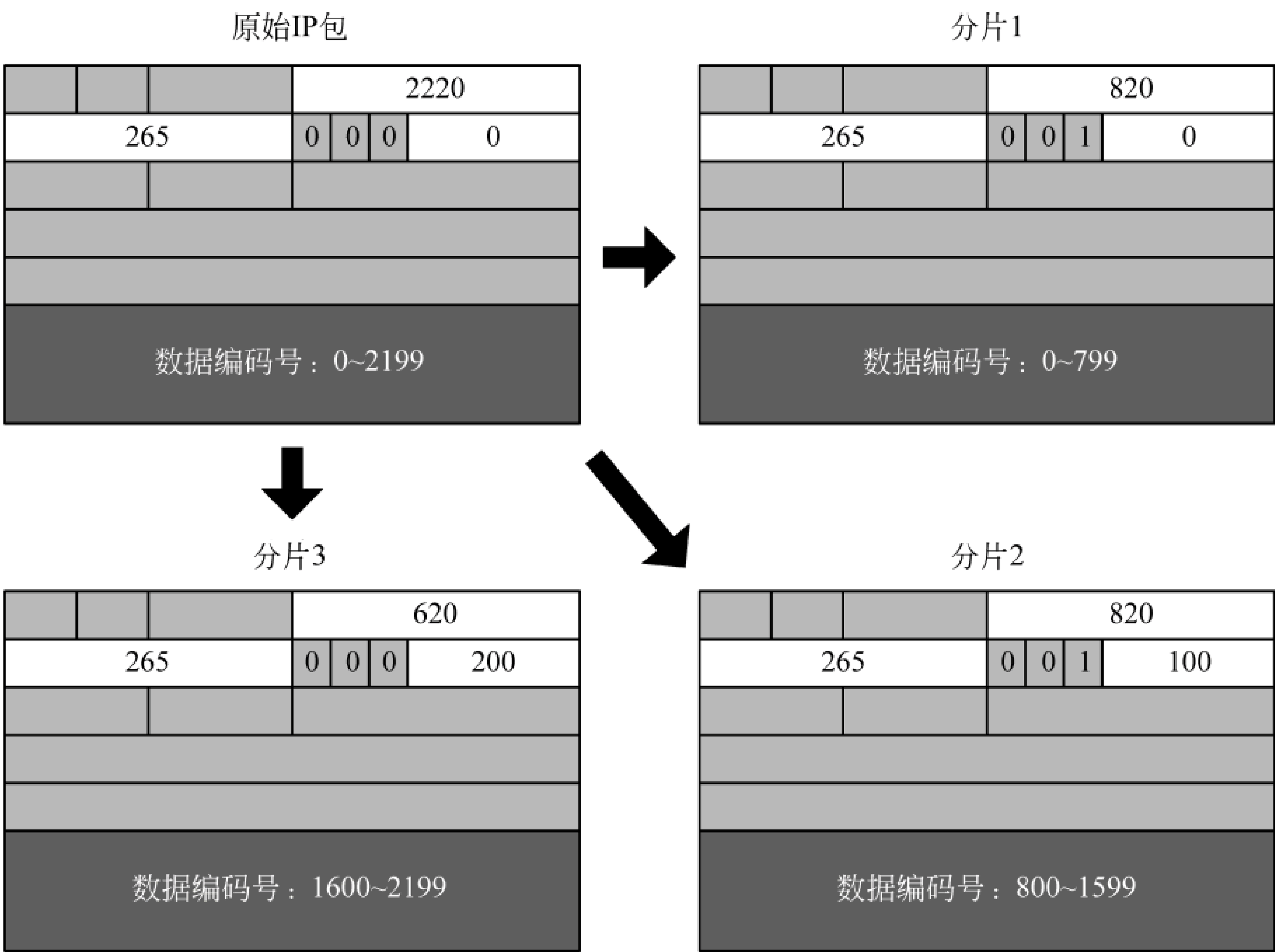


图 7-3　IP 包分片的相关字段值

从原始的 IP 数据包到对它进行分片后,IP 头部的总长度、标志位与片偏移等字段均发生了变化。特别是标志位中的 MF,分片 1 与分片 2 中的 MF 均为 1,表示不是最后一个分片;分片 3 中的 MF 为 0,表示是最后一个分片。需要注意的是,由于总长度、标志位与片偏移值都发生了变化,因此每个 IP 分片中的校验和需要重新计算。这里,IP 分片的校验和计算采用的是“二进制反码求和”算法。

7.3　例题分析

7.3.1　设计要求

根据 IPv4 协议规定的 IP 包的标准格式,编写程序来对现有的 IP 包进行分片,并将分

片后的 IP 头部与数据字段写入输出文件。数据字段值从指定的输入文件中获得。在本练习中为了简便起见，自行填写 IP 头部中除校验和外的各字段，以 80B 为单位对 IP 包进行分片。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 PackFrag.exe，则程序的命令行格式为：

```
PackFrag input_file output_file
```

其中，input_file 为输入文件，output_file 为输出文件。

(2) 要求将部分字段内容显示在控制台上，具体格式为：

```
IP包 1开始封装
总长度:xx
标识符:xx
标志位:xx,DF,MF
片偏移:xx
头部校验和:xx
数据字段:…
IP包 2开始封装
…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

7.3.2 关键问题

1. 填充分片头部字段

IP 包的分片采用的是与 IP 头部相同的结构。首先，需要构造一个 IP 头部的数据结构；然后，依次填充 IP 头部中的各个相应字段。这时，重点处理的是标志位与片偏移字段，这两个字段与 IP 包的分片、重组密切相关。如果某个分片不是最后一个分片，则将 3 位的标志位设置为 001；如果某个分片是最后一个分片，则将 3 位的标志位设置为 000。由于标志位与片偏移字段共同使用 16 位，因此需要通过移位以及与、或操作来完成。

下面给出设置标志位字段的伪代码：

```
if(数据长度<分片长度)
{
    bpacket=false;
    npacklen=nlength;
    ip.Flags=unsigned short((npacknum-1) * 80/8);
}
else
{
```

```
        nlength=nlength-80;
        npacklen=80;
        ip.Flags=unsigned short(((npacknum-1) * 80/8)|0x2000);
    }
```

2. 计算头部校验和函数

头部校验和字段的长度为 16 位(2B),用于保存检验 IP 头部是否出错的校验码。头部校验和的校验范围是整个 IP 头部。头部校验和的计算方法为:首先将头部校验和字段值置为 0,然后将 IP 头部值以 16 位为单位进行累加(异或),最后将累加结果取补码写入头部校验和字段。

下面给出计算头部校验和的伪代码:

```
unsigned short checksum(unsigned short * buffer,int size)
{
    unsigned long cksum=0;
    while(size>1)
    {
        cksum+= * buffer++;
        size-=sizeof(unsigned short);
    }
    if(size)
        cksum+= * (unsigned char * )buffer;
    cksum= (cksum>>16)+ (cksum&0xffff);
    cksum+= (cksum>>16);
    return(unsigned short) (~cksum);
}
```

3. 程序流程图

图 7-4 给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要有一个输入文件名与一个输出文件名。如果命令行参数的个数不是两个,则程序在输出错误信息后退出。在主程序的流程中,需要判断输入文件能否打开,以及 IP 包是否需要分片。

7.3.3 程序源代码

下面给出 IP 包分片封装程序的源代码:

```
//PackFrag.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "winsock2.h"
#include "fstream"
#include "iostream"
using namespace std;
```

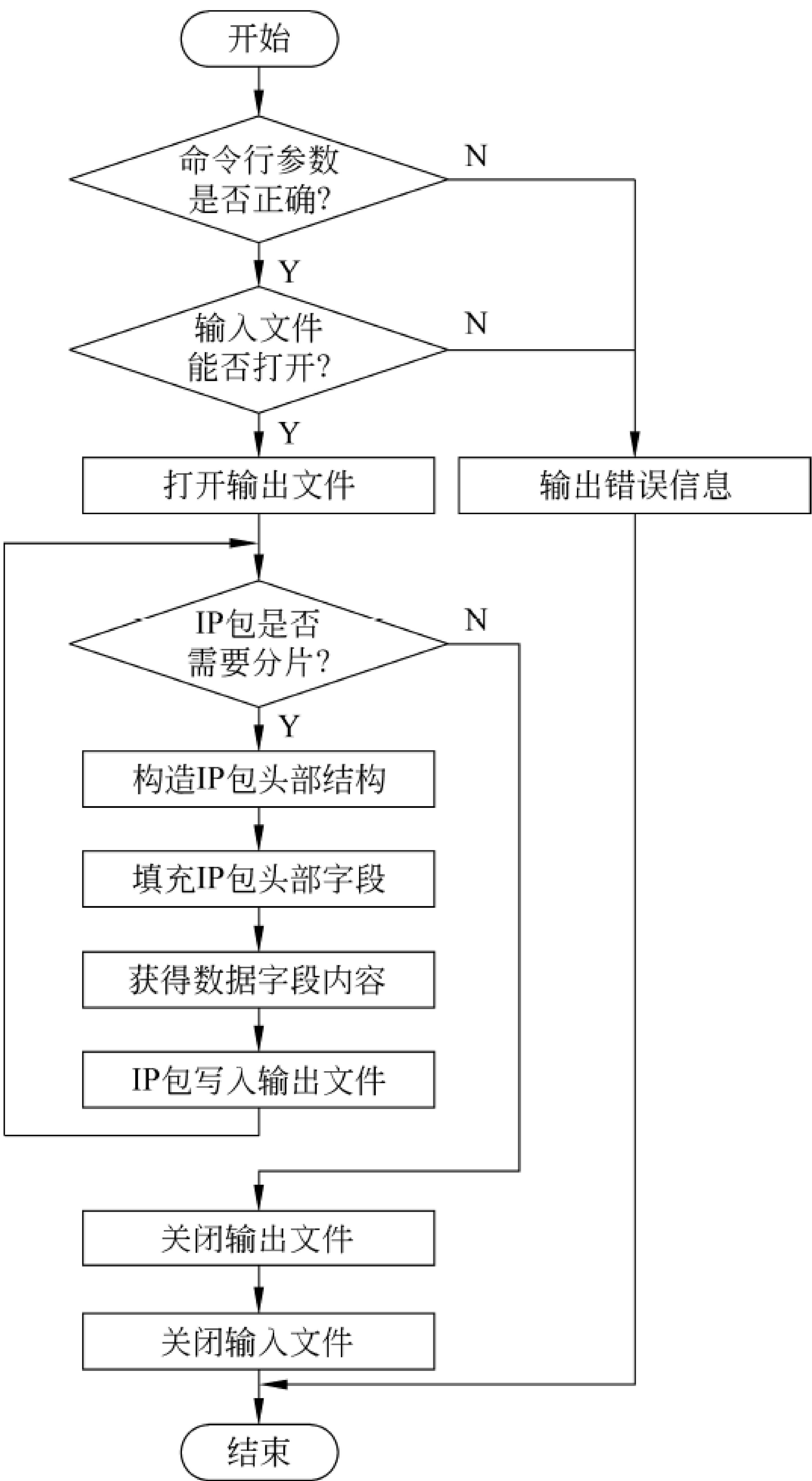


图 7-4　主程序流程图

```
#pragma comment(lib,"ws2_32")                     //加载 ws2_32.lib

typedef struct IP_HEAD                            //定义 IP 头部结构
{
    union
    {
        unsigned char Version;                    //版本(字节前 4 位)
        unsigned char HeadLen;                    //头部长度(字节后 4 位)
    };
    unsigned char ServiceType;                    //服务类型
    unsigned short TotalLen;                      //总长度
    unsigned short Identifier;                    //标识符
    union
    {
```

```
            unsigned short Flags;                  //标志位(字前 3 位)
            unsigned short FragOffset;             //片偏移(字后 13 位)
        };
        unsigned char TimeToLive;                  //生存周期
        unsigned char Protocol;                    //协议
        unsigned short HeadChecksum;               //头部校验和
        unsigned int SourceAddr;                   //源 IP 地址
        unsigned int DestinAddr;                   //目的 IP 地址
    }ip_head;

    unsigned short checksum(unsigned short * buffer,int size)
    {                                              //校验和计算函数
        unsigned long cksum=0;
        while(size>1)
        {
            cksum+=*buffer++;
            size-=sizeof(unsigned short);
        }
        if(size)
            cksum+=*(unsigned char*)buffer;
        cksum=(cksum>>16)+(cksum&0xffff);
        cksum+=(cksum>>16);
        return(unsigned short)(~cksum);
    }

    void main(int argc, char* argv[])
    {
        if(argc!=3)                                //检查命令行参数
        {
            cout<<endl<<"请按以下格式输入命令行:PackFrag input_file output_file"<<endl;
            return;
        }

        fstream outfile;                           //创建输出文件流
        outfile.open(argv[2],ios::out);            //打开输出文件
        fstream infile;                            //创建输入文件流
        infile.open(argv[1],ios::in);              //打开输入文件
        if(!infile.is_open())                      //输入文件能否打开
        {
            cout<<endl<<"无法打开输入文件"<<endl;
            return;
        }
```

```
infile.seekg(0,ios::end);                    //文件指针指向结尾
int nlength=infile.tellg();                  //获得输入文件长度
infile.seekg(0,ios::beg);                    //文件指针指向开始

bool bpacket=true;
int npacklen=0;
int npacknum=0;
unsigned short check[20];                    //设置校验缓冲区大小
while(bpacket)                               //开始分片封装 IP 包
{
    ip_head ip={0};
    npacknum++;

    cout<<endl<<"IP 包"<<npacknum<<"开始封装"<<endl;
    if(nlength<80)                           //判断 IP 包是否分片
    {
        bpacket=false;
        npacklen=nlength;
        ip.Flags=unsigned short((npacknum-1)*80/8);
    }
    else
    {
        nlength=nlength-80;
        npacklen=80;
        ip.Flags=unsigned short(((npacknum-1)*80/8)|0x2000);
    }

    //逐位写入 IP 头部各字段
    ip.Version=(0x04<<4|sizeof(ip_head)/sizeof(unsigned int));
    ip.ServiceType=unsigned char(0x00);
    ip.TotalLen=unsigned short(npacklen+20);
    ip.Identifier=unsigned short(0x1000);
    ip.TimeToLive=unsigned char(0x80);
    ip.Protocol=unsigned char(0x06);
    ip.HeadChecksum=unsigned short(0x00);
    ip.SourceAddr=unsigned int(0x3801a8c0);
    ip.DestinAddr=unsigned int(0x3502a8c0);

    memset(check,0,20);
    memcpy(check,&ip,20);
    ip.HeadChecksum=checksum(check,20);      //计算 IP 头部校验和

    cout<<"总长度:"<<ip.TotalLen<<endl;
    cout<<"标识符:"<<ip.Identifier<<endl;
```

```
            cout<<"标志位:"<<((ip.Flags>>15)&0x01)<<",DF="<<((ip.Flags>>14)&0x01)<<",
            MF="<<((ip.Flags>>13)&0x01)<<endl;
            cout<<"片偏移:"<<(ip.FragOffset&0x1fff)<<"(8B)"<<endl;
            cout<<"头部校验和:"<<ip.HeadChecksum<<endl;
            outfile.write((char*)&ip,20);            //写入 20B 的 IP 头部

            char* data=new char[npacklen];
            infile.read(data,npacklen);              //输入文件读出数据
            cout<<"数据字段:";
            for(int i=0;i<npacklen;i++)
                cout<<data[i];
            cout<<endl;
            outfile.write(data,npacklen);            //数据写入输出文件
            delete data;
        }

        cout<<endl<<"IP 包分片封装完成"<<endl;
        outfile.close();                             //关闭输出文件
        infile.close();                              //关闭输入文件
    }
```

图 7-5 给出了 IP 包分片封装的过程。程序命令行输入为 PackFrag input output。程序首先依次填写每个 IP 分片头部的各个字段,并将 input 中的数据拆分后封装到相应分片的数据部分,然后将每个 IP 分片的内容依次写入 output 中。

```
命令提示符
E:\Test\PackFrag\Debug>PackFrag

请按以下格式输入命令行: PackFrag input_file output_file

E:\Test\PackFrag\Debug>PackFrag input output

IP包1开始封装
总长度: 100
标识符: 4096
标志位: 0, DF=0, MF=1
片偏移: 0 (8B)
头部校验和: 2642
数据字段: Nankai University was founded in 1919 by the famous patriotic educator
s in Chine

IP包2开始封装
总长度: 72
标识符: 4096
标志位: 0, DF=0, MF=0
片偏移: 10 (8B)
头部校验和: 10852
数据字段: se modern history, Mr. Zhang Boling and Mr. Yan Xiu.

IP包分片封装完成
```

图 7-5　IP 包分片封装的过程

7.4 练　习　题

根据 IPv4 协议规定的 IP 包的标准格式，编写程序对分片后的多个 IP 包进行重组，并将重组后的 IP 包的数据字段写入输出文件。在本练习中为了简便起见，分片后的多个 IP 包从指定的输入文件中获得，重组后的 IP 头部需要重新计算校验和。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 PackEncap.exe，则程序的命令行格式为：

```
PackEncap input_file output_file
```

其中，input_file 为输入文件，output_file 为输出文件。

(2) 要求将部分字段内容显示在控制台上，具体格式为：

```
总长度:xx
标识符:xx
标志位:xx,DF,MF
片偏移:xx
头部校验和:xx
数据字段:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 8 章 IPv6 数据包的封装与解析

8.1 设 计 目 的

IPv6 协议是针对当前 IP 协议问题制订的下一代协议标准。熟悉 IPv6 协议对于理解网络层协议的概念、层次结构与执行过程有很大的帮助。本章练习的目的是根据 IPv6 包的标准格式，通过封装与解析 IPv6 包来了解 IPv6 协议的基本概念，从而深入理解下一代网络层协议的工作过程。

8.2 相 关 知 识

本章涉及的相关知识包括 IPv4 协议的主要缺点、IPv6 协议的概念与 IPv6 数据包结构，以及 IPv6 地址结构等。

8.2.1 IPv4 协议的主要缺点

随着 Internet 的规模的扩大与应用的深入，作为 Internet 核心协议之一的 IPv4 协议一直处于不断补充、完善和提高的过程中，但是 IPv4 版本的主要内容没有发生任何实质性的变化。IP 协议能够适应复杂的应用需求，对推动 Internet 发展起到了重要的作用。实践证明，IPv4 协议是健壮和易于实现的，并且具有很好的互操作性。它本身也经受住了 Internet 从小型的科研范围应用的互联网络，发展成全球性的大规模网际网的考验，这些都说明 IPv4 协议的设计是成功的。

IP 协议是一种无连接、不可靠的协议，是在异构网络中提供分组传送服务的协议。IP 协议的设计思想就是通过简单的方法解决复杂问题，采用“尽力而为”的服务去应对互联网络中存在的各种复杂问题。这是 IPv4 协议的成功之处，同样也是 IPv4 掣肘的地方。随着计算机网络规模的不断扩大，IPv4 协议也逐渐暴露出它的缺陷。20 世纪 80 年代初就已经开始研究的 IPv4 协议，在今天看出问题也是很自然的事，近年来，研究人员针对这些问题不断提出各种改进意见。

IPv4 存在的问题主要表现在以下几个方面。

（1）标准分类地址的利用率低，地址数量不能满足网络规模不断扩展的需要。

（2）随着网络结构越来越复杂，路由选择算法的研究越来越显得困难。

(3) IPv4 协议对分组传输可靠性没有提供任何保障措施。

(4) IPv4 协议不支持多播传输。

(5) IPv4 协议不能保证分组传输的服务质量。

(6) IPv4 协议对网络安全问题没有提出对策。

IPv4 协议发展过程可以从不变和变化的两方面来认识。IPv4 协议中对于分组结构与分组头部结构的基本定义是不变的。变化的部分主要集中在三方面：①IP 地址处理方法；②分组交付需要的路由算法与路由协议；③为提高协议的可靠性、服务能力与安全性增加的补充协议。任何事情都有一个限度。随着网络规模的继续扩大与应用的不断深入，当这些补充协议已经无法从根本上解决问题时，就需要彻底考虑、重新设计新的协议，这就导致了 IPv6 协议的研究与应用。

8.2.2 IPv6 协议的基本概念

IPv4 协议的设计者无法预见 20 年来 Internet 技术发展得如此之快，Internet 应用会变得如此广泛。IPv4 协议面对的很多问题已经无法通过打“补丁”的方法解决，只能在设计新一代 IP 协议时统一加以考虑并解决。针对这种情况，IETF 设计了一套全新的协议标准——IPv6。实际上，IPv6 是由多个层次的一系列相关协议所构成的协议集。IPv6 协议在设计中尽量做到对上层、下层协议的影响最小，并力求在协议设计时考虑得更为周全，以避免后续仍需不断做出新的修补与改进。

1992 年，IETF 成立了专门的 IPng 工作组，致力于下一代 IP 协议的制订。1994 年，IPng 工作组公布了 RFC1726 文档，提出了关于下一代 IP 协议的 18 个选择方案。1995 年，Cisco 公司与 Nokia 公司的研究人员起草了 IPv6 协议的最初草案。1996 年，IETF 启动了建立全球 IPv6 实验床 6Bone。1998 年，IETF 正式公布了 IPv6 协议标准——RFC2460。1999 年，IETF 成立了 IPv6 论坛，正式开始分配 IPv6 地址。2001 年，主流的操作系统(Windows、Linux、Solaris 等)开始支持 IPv6 协议。2003 年，主要的网络设备制造商开始提供 IPv6 设备。同年，我国启动中国下一代互联网示范工程(CNGI)建设。

IPv6 协议的特点主要表现在以下几个方面。

1. 新的协议头部格式

IPv6 头部采用了一种全新的数据格式，IPv4 头部长度是可变的，而 IPv6 基本头部长度是固定的，将一些非根本性与可选的字段移到扩展头部，并且仅有“逐跳”头部需要由转发的路由器来处理，这样使路由器在处理协议头部时效率更高。

2. 巨大的地址空间

IPv6 地址长度从 IPv4 的 32 位增大到 128 位，使 IPv6 可以提供多达超过 3.4×10^{38} 个 IP 地址，为未来接入移动互联网、物联网的移动设备提供更多的 IP 地址。IPv6 协议可以从根本上解决 IP 地址匮乏问题，不再使用带来很多问题的 NAT 技术。

3. 有效的分层路由结构

IPv6 更大的地址空间能更好地将路由结构划分出层次，可以覆盖从各级主干网直到内部子网的多级结构。IPv6 将分配给主机的 128 位地址分为两部分，其中 64 位可作为子网地址空间来使用，另外 64 位用于映射相关网卡硬件地址。

4. 灵活的地址自动配置

为了简化主机与路由器的网络配置过程,IPv6支持两种地址自动配置方法:有状态地址自动配置与无状态地址自动配置。IPv6地址自动配置方法使主机在接入IPv6网络后,在无须用户干预的情况下自动获得可用的IP地址。

5. 内置的安全性服务

IPSec协议作为一个IPv6的组成部分而使用。IPSec提供认证头部与封装安全载荷两种子协议,以及用来处理安全设置的密钥交换协议,它可以提供主机IP地址认证、数据完整性验证与数据加密等功能。

6. 更好地支持QoS

IPv6头部中的流标记字段定义如何识别通信流,路由器可对属于一个流的数据包进行特殊处理。由于通信流是在IPv6基本头部中加以标识,因此即使对IP数据包执行IPSec的加密操作,仍然能够方便地实现对QoS的支持。

7. 良好的可扩展性

IPv6支持在基本头部之后定义自己的扩展头部,可以方便地实现对新增加的网络应用的支持与扩展。IPv4头部最多只能支持40B的选项,IPv6扩展头部长度只受IPv6包长度的限制。

8.2.3 IPv6数据包的结构

IPv6协议极大地简化了IP基本头部的结构,并将所有非核心功能都交给扩展头部实现。RFC1883是最早出现的IPv6协议文档,它描述了IPv6协议的基本内容,主要是IPv6数据包的基本结构。RFC2460是对RFC1883文档的更新版本。后来,出现了几十种对IPv6协议进行扩展与调整的RFC文档。IPv6数据包由三个部分组成:基本头部、扩展头部与数据部分。图8-1给出了IPv6数据包的基本结构。其中,基本头部是长度固定为40B的必备部分;扩展头部是可供选择的多种用途头部的集合;数据部分可能包括传输层、网络层或应用层的协议数据。

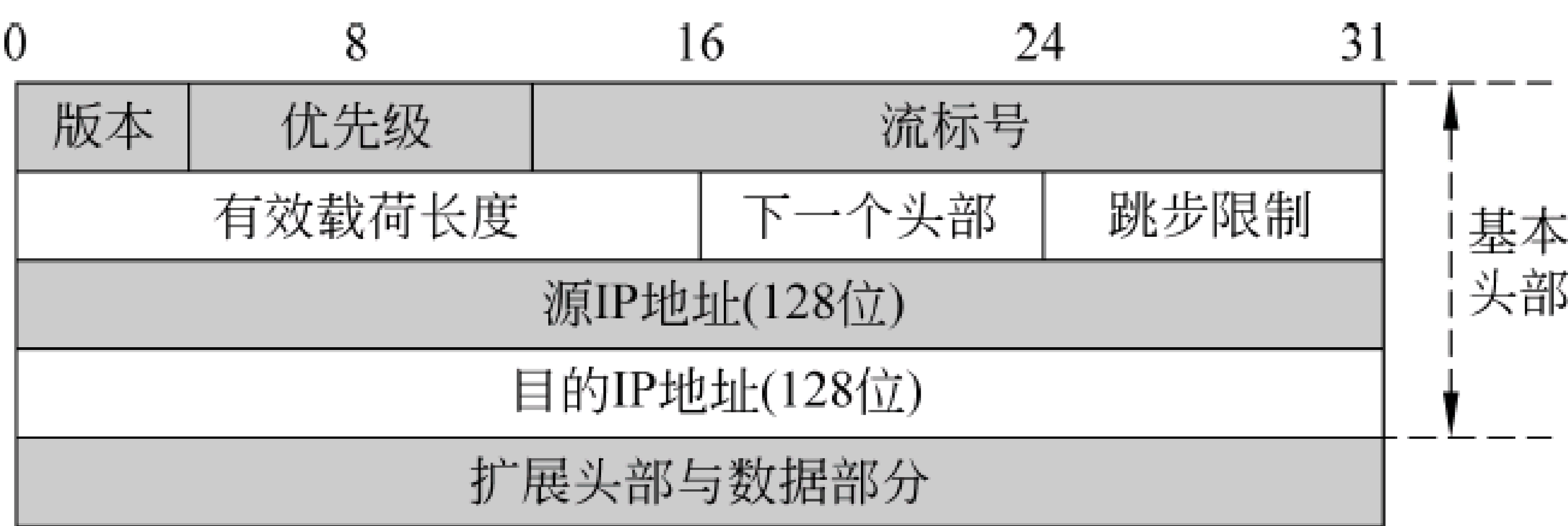


图8-1 IPv6数据包的基本结构

1. 基本头部

IPv6基本头部由以下字段组成。

1) 版本

版本(version)的长度为4位,表示数据包使用的IP协议版本。这里,IPv6协议的版本号字段值为6。

2）优先级

优先级(priority)的长度为 8 位，表示数据包的类型与优先级，路由器通过它决定在网络拥塞时如何处理该数据包。例如，优先级字段值为 0～7 时，表示在拥塞发生时允许延时处理；优先级字段值为 8～15 时，表示优先级较高的实时业务需要使用固定速率来传输。优先级字段的默认值是 0，表示不使用区分服务。在 RFC2460 中，对优先级字段的使用没有明确定义。

3）流标号

流标号(flow label)的长度为 20 位，表示数据包属于源主机与目的主机之间的某个数据流，它需要由中间的 IPv6 路由器进行特殊处理。流标号用于非默认的 QoS 连接，例如实时数据(音频与视频)的连接。源主机与目的主机之间可能有多个数据流，它们需要用不同的流标号来加以区别。流标号字段的默认值是 0，表示不使用区分服务。在 RFC2460 中，对流标号字段的使用没有明确定义。

4）有效载荷长度

有效载荷长度(payload length)的长度为 16 位，表示数据包中除了基本头部之外的数据长度，这部分包括扩展头部与高层协议数据。IPv6 数据包的有效载荷部分的最大长度为 65 535B。

5）下一个头部

下一个头部(next header)的长度为 8 位，表示位于基本头部后面的数据类型。这个字段的功能类似于 IPv4 的协议字段。如果 IPv6 数据包中存在扩展头部，该字段标识紧跟扩展头部类型，例如逐跳头部、路由头部等类型；否则，该字段标识上层协议类型，例如传输层协议(TCP 或 UDP)或网络层协议(ICMP)类型。

6）跳步限制

跳步限制(hop limit)的长度为 8 位，表示数据包可以通过的最大的路由器转发次数。这个字段的功能类似于 IPv4 的生存时间，防止数据包因出现问题而在网络中无限转发。数据包每经过一个路由器，该字段的值减 1。当该字段为 0 时，路由器丢弃该数据包，并向源主机发送相应的 ICMPv6 报文。

7）IP 地址

IP 地址(IP address)包括两个部分：源 IP 地址与目的 IP 地址。这里，源 IP 地址与目的 IP 地址的长度均为 128 位。源 IP 地址是发送数据包的源主机 IPv6 地址；目的 IP 地址是接收数据包的目的主机 IPv6 地址。

2. 扩展头部

IPv6 扩展头部是用来扩展协议功能的部分。目前，IPv6 协议已经定义了 7 种扩展头部。表 8-1 给出了主要的 IPv6 扩展头部。每种扩展头部在下一个头部字段中对应不同值。每种扩展头部的格式与长度都各不相同，但扩展头部的长度必须是 8B 的整数倍。如果 IPv6 数据包中包含多个扩展头部，这些扩展头部将会形成一种链状结构。IPv6 扩展头部的排列顺序依次是：逐跳头部、目的地选项头部、路由头部、分片头部、认证头部、封装安全载荷头部。除了目的地选项头部之外，其他扩展头部在 IPv6 数据包中只能出现一次。RFC2460 没有对涉及安全的认证头部与封装安全载荷头部进行详细说明。

表 8-1 主要的 IPv6 扩展头部

下一个头部字段值	IPv6 扩展头部名称
0	逐跳头部(hop-by-hop header)
43	路由头部(routing header)
44	分片头部(fragment header)
51	认证头部(authentication header)
52	封装安全载荷头部(ESP header)
60	目的地选项头部(destination options header)

IPv6 基本头部与扩展头部中都有下一个头部字段,其数值用来指出下一个头部的类型,可以是 IPv6 扩展头部或上层协议头部。图 8-2 给出了带扩展头部的 IPv6 数据包结构。例如,在基本头部中,下一个头部字段值为 44,则后面紧跟的是分片头部;在分片头部中,下一个头部字段值为 51,则后面紧跟的是认证头部。如果认证头部是最后一个扩展头部,则认证头部中的下一个头部字段值应该为 2、6 或 17,这三个值表示不同的上层协议(分别为 ICMP、TCP 或 UDP 协议)。如果 IPv6 数据包中不包含扩展头部,也不打算填充高层数据,则基本头部中的下一个头部字段值应该为 59。

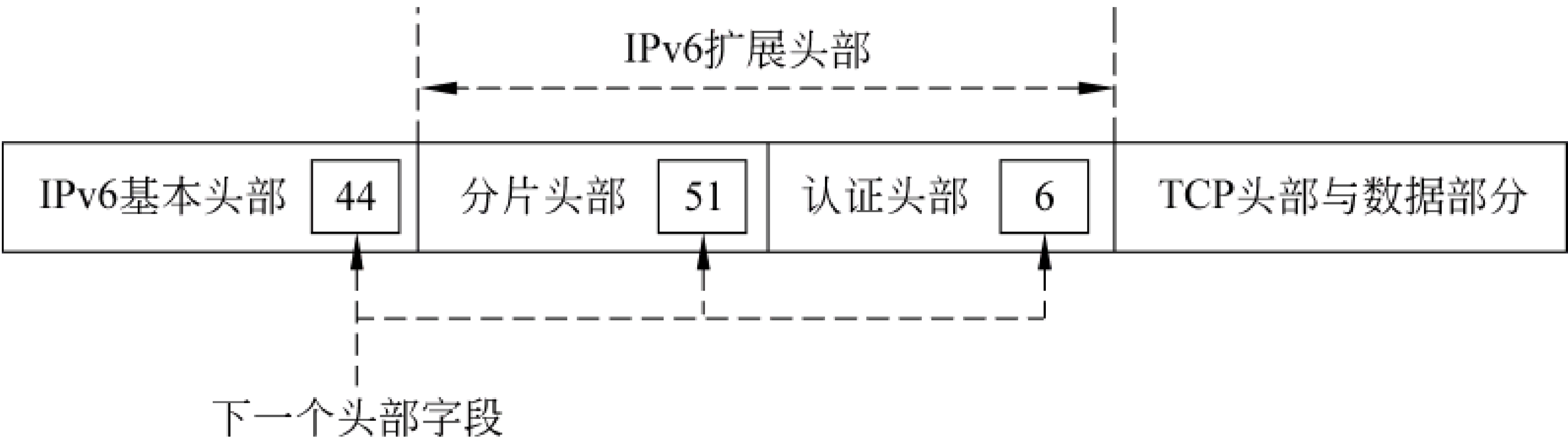


图 8-2 带扩展头部的 IPv6 数据包结构

逐跳头部携带需要路由器特殊处理的信息;目的地选项头部携带只能由目的主机检查的信息;路由头部的作用类似于 IPv4 的源路由选项;分片头部表示源主机发送大于最大传输单元长度的数据包。一般的 IPv6 数据包并不需要这么多的扩展头部,只是在转发路由器或目的主机配合做一些特殊处理时才需要这些扩展头部。例如,网络软件测试与网络故障诊断,源主机才会添加一个或几个必要的扩展头部。每个转发 IPv6 数据包的中间路由器,仅处理固定长度的基本头部,而唯一要处理的扩展头部是逐跳头部。因此,这种做法必然会提高路由器处理 IPv6 头部的速度,缩短路由器转发 IPv6 数据包的延迟时间。

8.2.4 IPv6 地址结构

IPv6 地址的长度由 IPv4 地址的 32 位增加到 128 位。IPv6 地址结构由三部分组成:地址前缀、接口标识与中间部分。图 8-3 给出了 IPv6 地址的基本结构。其中,地址前缀是位于 IPv6 地址最高位的地址区分部分,不同类型地址的地址前缀的长度与内容均不同;接口标识是位于 IPv6 地址最低位的 64 位,采用 64 位的 EUI-64 格式的 MAC 地址;中间部分是位于地址前缀与接口标识之间的部分,用来构成 IPv6 地址中用于路由的层次结构。



图 8-3 IPv6 地址的基本结构

RFC3513 文档规定了 IPv6 地址的基本结构。

IPv6 地址的长度为 128 位，采用"冒号分十六进制"方式，即 $x:x:x:x:x:x:x:x$ 的格式表示，每个地址段 x 为 16 位。例如，ABCD:EF01:2345:6789:ABCD:EF01:2345:6789 是一个合法的 IPv6 地址。在这种表示法中，每个地址字段内数值前面的 0 都可省略，但是每个地址段中至少应该有一个数值。某些 IPv6 地址可能包含长串的 0，为了便于以文本方式描述这种地址，可用双冒号"::"符号来表示 1 组或多组 16 位 0，但是"::"只能在一个地址中出现一次。例如，FEC0:1:0:0:0:0:0:1234 可以表示为 FEC0:1::1234。128 位的 IPv6 的地址实在太长，人们很难记忆。在 IPv6 网络中，所有 IPv6 地址都是自动配置的。

IPv6 地址不再采用子网掩码表示方法，它只支持前缀长度表示法。前缀是 IPv6 地址的一个部分，用作 IPv6 路由或子网标识。IPv6 前缀可以用"地址/前缀长度"来表示。如果一个节点的 IPv6 地址为 2001:FA2:0:FE08::9C5A，地址前缀长度为 48 位，则节点的子网号为 2001:FA2::/48。前缀 48 位表示地址的前 48 位为网络地址，之后的 80 位可分配给网络中的主机，可以分配给主机的地址数量共有 2^{80} 个。IPv6 地址空间能更好地对路由结构划分层次，覆盖从主干网到内部子网的多级结构，IPv6 地址可以按照具体用途进行分类。

由于 IPv6 地址一直处于改进过程中，因此在发现问题之后一定会有新的 RFC 发布，用来修订 IPv6 地址分配方案。1998 年发布的 RFC2373 是最早的方案，2003 年发布的 RFC3513 是一个修订方案，2006 年发布的 RFC4291 是最新的方案。例如，本地站点地址(site-local address)类似于 IPv4 中的专用地址，本地站点地址出现在公网上，有可能带来一定的安全问题，因此被废除。总之，研究 IPv6 地址需要密切注意新的 RFC 文档的发布，关于 IPv6 地址问题的 RFC 文档很多，并且更新速度很快。

目前，IPv6 地址可以分为三种基本类型：单播地址、组播地址和任播地址。其中，单播地址(unicast address)用来标识路由器、主机的某个网络接口，发送到单播地址的 IPv6 数据包，被交付给该地址标识的网络接口。组播地址(multicast address)用来标识一组属于不同节点的网络接口，发送到多播地址的 IPv6 数据包，被交付给由该地址标识的所有网络接口。任播地址(anycast address)用来标识一组属于不同路由器的网络接口，发送到任播地址的 IPv6 数据包，被交付给由该地址标识的一组接口中距离"最近"的一个。IPv6 协议不使用广播地址，广播地址的功能由组播地址所代替。

单播 IPv6 地址是最重要的一类 IPv6 地址，主要包括三种不同用途的单播地址，图 8-4 给出了单播 IPv6 地址的基本结构。可汇聚全球单播地址(aggregatable global unicast address)可在全球范围内 IPv6 网络中提供有效的路由和转发。它可以支持三层的网络拓

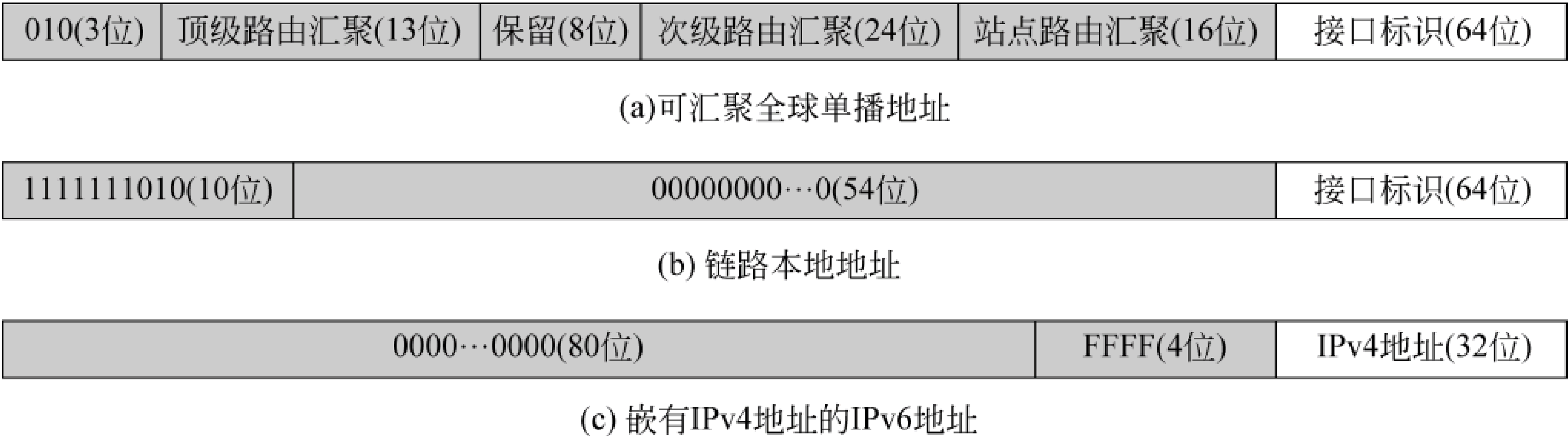


图 8-4　单播 IPv6 地址的基本结构

扑结构:顶级路由汇聚、次级路由汇聚与站点路由汇聚。链路本地地址(link-local address)用于同一链路上的相邻节点之间的通信,路由器不转发带有链路本地地址的IPv6数据包。嵌有IPv4地址的IPv6地址(IPv6 addresses with embedded IPv4 addresses)又称为IPv4映射地址,它是应用在IPv6与IPv4共存阶段的特殊地址。

8.2.5 IPv6安全功能

IPSec(Internet Protocol Security)是IETF针对网络层通信安全而制订的一个协议集,目前已经被广泛应用于VPN系统中。IPSec适用于各个IP协议版本(IPv4与IPv6),IPv4将IPSec作为一种可选的扩展协议,而IPv6将它作为一个组成部分来使用。IPSec的设计目标是为IP分组传输提供辅助安全服务。例如,数据源身份认证、数据完整性认证与数据加密等。1995年,RFC1825～RFC1829定义了IPSec。由于IPSec是工作在网络层的安全协议,因此任何上层协议都可以使用IPSec提供的安全服务。

IPSec主要包括三个组成部分:认证头部(Authentication Header,AH)、封装安全负载(Encapsulating Security Payload,ESP)与密钥管理协议。其中,AH协议可提供数据源身份认证、数据完整性认证,以及可选的抗重放数据包功能;ESP协议可提供AH协议的所有功能与数据加密服务;密钥管理协议用于通信双方之间协商安全参数,例如工作模式、认证或加密算法、密钥与生存期等。实际上,AH与ESP协议都是网络层的安全协议,而密钥管理协议是应用层的安全协议。

IPSec还包括以下几个部分:IPSec安全结构、解释域、认证算法与加密算法。其中,IPSec安全结构是IPSec的总体框架结构,它是理解整个IPSec协议集的基础;解释域将所有IPSec相关文献绑定起来,它是所有IPSec安全参数的主数据库,这些参数能被使用IPSec服务的系统调用;认证算法是可供AH与ESP选择的认证算法,例如MD5、SHA-1等算法;加密算法是可供ESP协议选择的加密算法,例如DES算法。IPSec协议集已经确定上述算法可用。另外,IPSec可以通过RFC增加其他可用的算法。

1999年,IETF对IPSec进行了较大范围的改进,由RFC2401～RFC2412代替原有的RFC。在本次改进中,主要增加了几种密钥管理协议:互联网安全关联与密钥管理协议(Internet Security Association and Key Management Protocol,ISAKMP)、互联网密钥交换(Internet Key Exchange,IKE)与Oakley等。这些密钥管理协议支持自动建立安全连接,以及自动分发与更新密钥等功能。IPSec通过IKE完成安全协议的安全参数协商。这里,安全参数是安全协议相关的密钥、生存期和发布方式等。

8.3 例题分析

8.3.1 设计要求

根据协议规定的IPv6数据包的标准格式,编写程序构造IPv6包结构(包括IPv6头部与TCP头部),然后将封装后的IPv6包内容写入输出文件。在本练习中为了简便起见,不需要构造任何IPv6扩展头部,数据字段通过为字符串赋值来获得,但是需要计算TCP头部与数据部分的校验和。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 Ipv6Encap.exe,则程序的命令行格式为:

```
Ipv6Encap output_file
```

其中,output_file 为输出文件。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
IP 头部与数据字段
版本:xx
有效载荷长度:xx
下一个头部:xx
源 IP 地址:xx:xx:xx:xx:xx:xx:xx:xx
目的 IP 地址:xx:xx:xx:xx:xx:xx:xx:xx
数据字段:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

8.3.2 关键问题

1. 定义 IPv6 头部的数据结构

在对 IPv6 包的各字段进行填充之前,首先需要构造一个 IPv6 头部数据结构。这个数据结构要与图 8-1 的 IPv6 头部结构一致。IPv6 地址由两个部分来构造:64 位的前缀与 64 位的 EUI-64 接口标识。另外,需要构造 TCP 头部与伪头部数据结构,伪头部只是用来为 TCP 包计算校验和,并不需要作为 IPv6 包的一部分进行封装。

下面给出构造 IPv6 头部结构的伪代码:

```
//定义 IPv6 头部结构
typedef struct IP_HEAD
{
    union
    {
        unsigned int Version;
        unsigned int Priority;
        unsigned int FlowLabel;
    };
    unsigned short PayloadLen;
    unsigned char NextHead;
    unsigned char HopLimit;
    //源 IPv6 地址
    struct
    {
```

```
            __int64 Prefix;
            unsigned char MacAddr[8];
        }SourceAddr;
        //目的 IPv6 地址
        struct
        {
            __int64 Prefix;
            unsigned char MacAddr[8];
        }DestinAddr;
    }ip_head;
```

2. 填充 IPv6 包的各个字段

在对 IPv6 包的内容进行封装之前，需要分别填充 IPv6 头部、TCP 头部与 TCP 数据。在填充 IPv6 地址的过程中，前缀部分按全球单播地址标准格式填充，源 IP 地址的接口标识由 00-00-80-1A-E6-65 生成，目的 IP 地址的接口标识由 00-00-E4-86-3A-DC 生成。需要注意，标准 MAC 地址应该转换为 EUI-64 格式的 MAC 地址，只需在标准 MAC 地址的中间添加 FF-FE 即可，例如 00-00-80-FF-FE-1A-E6-65。

下面给出生成 IPv6 地址的伪代码：

```
//填充 3 位地址前缀
ip.SourceAddr.Prefix=0x1;
//填充 45 位路由前缀
ip.SourceAddr.Prefix<<=45;
ip.SourceAddr.Prefix+=0x01;
//填充 16 位子网号
ip.SourceAddr.Prefix<<=16;
ip.SourceAddr.Prefix+=0x01;
ip.DestinAddr.Prefix=hton64(ip.DestinAddr.Prefix);
//填充 8 字节 EUI-64 地址
ip.SourceAddr.MacAddr[0]=char(0x00);
ip.SourceAddr.MacAddr[1]=char(0x00);
ip.SourceAddr.MacAddr[2]=char(0x80);
ip.SourceAddr.MacAddr[3]=char(0xFF);
ip.SourceAddr.MacAddr[4]=char(0xFE);
ip.SourceAddr.MacAddr[5]=char(0x18);
ip.SourceAddr.MacAddr[6]=char(0x6E);
ip.SourceAddr.MacAddr[7]=char(0xE5);
```

另外，为了填充 TCP 头部的校验和字段，还需要填充 TCP 头部附带的伪头部。这时，首先为 TCP 头部的校验和字段赋初值 0，调用 checksum()函数对 TCP 头部、伪头部与数据部分进行计算，然后将获得的校验和的值填入校验和字段。注意，填充伪头部只是为了计算校验和，并不需要将伪头部内容写入输出文件。

3. 程序流程图

图 8-5 给出了主程序流程图。要求输入的命令行参数必须正确，除了程序本身的名称

以外，还需要有一个输出文件名。如果命令行参数的个数不是一个，则程序在输出错误信息后退出。

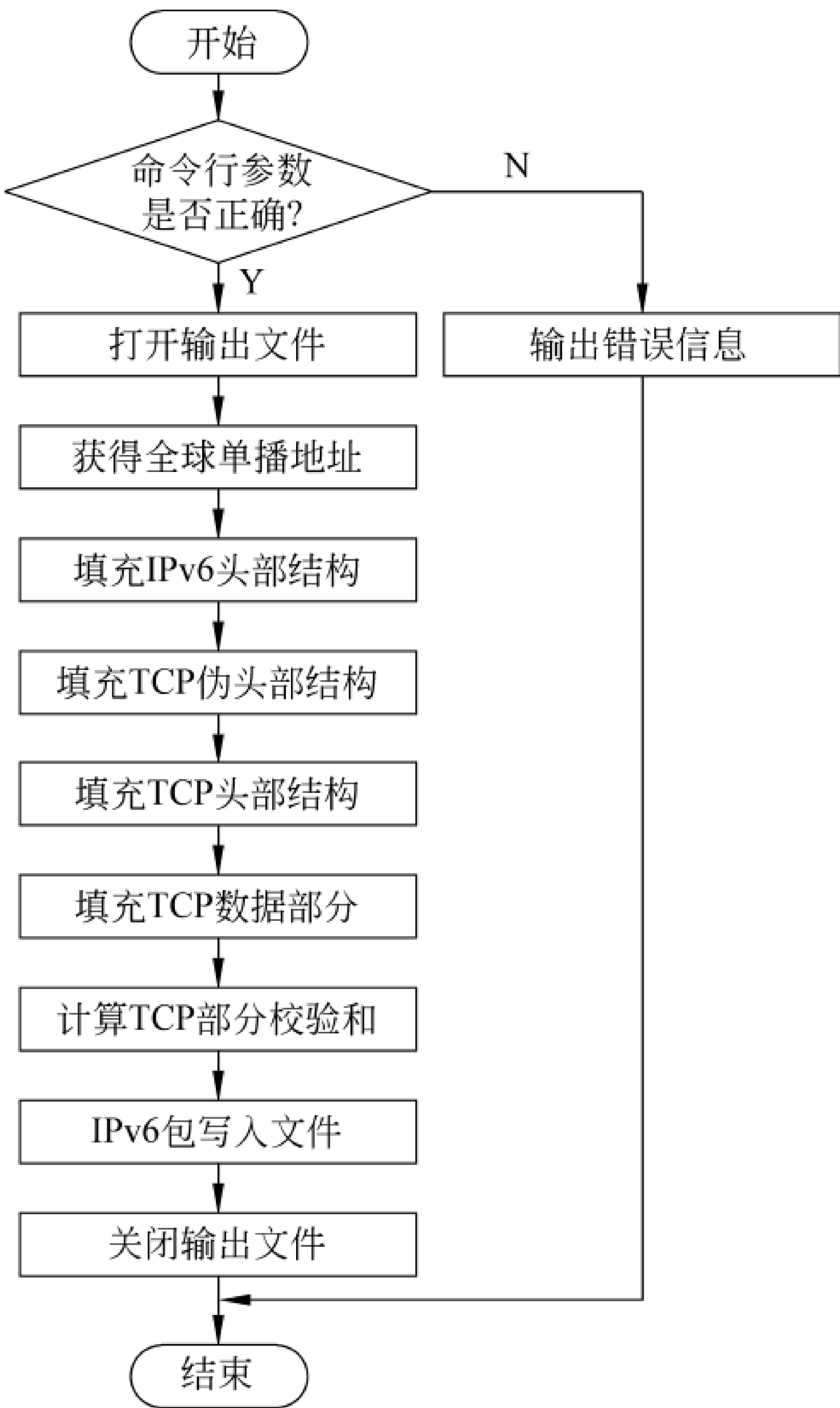


图 8-5　主程序流程图

8.3.3　程序源代码

下面给出 IPv6 包封装程序的源代码：

```
//Ipv6Encap.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "winsock2.h"
#include "fstream"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                        //加载 ws2_32.lib

typedef struct IP_HEAD                               //定义 IP 头部结构
{
```

```
    union
    {
        unsigned int Version;                   //版本 (双字前 4 位)
        unsigned int Priority;                  //优先级 (双字中 8 位)
        unsigned int FlowLabel;                 //流标号 (双字后 20 位)
    };
    unsigned short PayloadLen;                  //有效载荷长度
    unsigned char NextHead;                     //下一个头部
    unsigned char HopLimit;                     //跳步限制
    struct
    {
        __int64 Prefix;                         //前缀与子网号 (64 位)
        unsigned char MacAddr[8];               //主机 MAC 地址 (64 位)
    }SourceAddr;                                //源 IP 地址
    struct
    {
        __int64 Prefix;                         //前缀与子网号 (64 位)
        unsigned char MacAddr[8];               //主机 MAC 地址 (64 位)
    }DestinAddr;                                //目的 IP 地址
}ip_head;

typedef struct PSD_HEAD                         //定义 TCP 伪头部结构
{
    unsigned char SourceAddr[16];               //源 IP 地址
    unsigned char DestinAddr[16];               //目的 IP 地址
    unsigned char Reserved;                     //保留位
    unsigned char Protocol;                     //协议
    unsigned short TcpLen;                      //TCP 长度
}psd_head;

typedef struct TCP_HEAD                         //定义 TCP 头部结构
{
    unsigned short SourcePort;                  //源端口
    unsigned short DestinPort;                  //目的端口
    unsigned int Sequence;                      //序列号
    unsigned int Acknowledge;                   //确认号
    union
    {
        unsigned short HeadLen;                 //头部长度 (字前 4 位)
        unsigned short Reserved;                //保留位 (字中 6 位)
        unsigned short Flags;                   //标志位 (字后 6 位)
    };
    unsigned short WindowsLen;                  //窗口大小
    unsigned short TcpChecksum;                 //TCP 校验和
```

```
    unsigned short UrgePoint;                   //紧急指针
}tcp_head;

unsigned short check[65535];                    //设置校验缓冲区
const char tcp_data[]={"This is a test of ipv6 packet encapsule!"};

unsigned short checksum(unsigned short * buffer,int size)
{                                               //校验和计算函数
    unsigned long cksum=0;
    while(size>1)
    {
        cksum+= * buffer++;
        size-=sizeof(unsigned short);
    }
    if(size)
        cksum+= * (unsigned char * )buffer;
    cksum=(cksum>>16)+(cksum&0xffff);
    cksum+=(cksum>>16);
    return(unsigned short)(~cksum);
}

__int64 hton64(__int64 host64)                  //64位的字节序转换函数
{
    char temp;
    char * p=(char * )&host64;
    for(int i=0;i<4;i++)
    {
        temp=p[i];
        p[i]=p[7-i];
        p[7-i]=temp;
    }
    return host64;
}

void main(int argc,char * argv[])
{
    if(argc!=2)                                 //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:Ipv6Encap output_file"<<endl;
        return;
    }

    fstream outfile;                            //创建输出文件
    outfile.open(argv[1],ios::in|ios::out);     //打开输出文件
```

```
//填充 IP 包头部各字段
ip_head ip={0};
ip.Version=6;                                    //填充 4 位版本
ip.Version<<=8;
ip.Version+=0;                                   //填充 8 位优先级
ip.Version<<=20;
ip.Version+=0;                                   //填充 20 位流标号
ip.Version=htonl(ip.Version);
ip.PayloadLen=htons(sizeof(ip_head)+sizeof(tcp_head)+sizeof(tcp_data));
ip.NextHead=IPPROTO_TCP;                         //填充 8 位下一个头部
ip.HopLimit=128;                                 //填充 8 位跳步限制

//填充源 IP 地址
ip.SourceAddr.Prefix=0x1;                        //填充 3 位地址前缀
ip.SourceAddr.Prefix<<=45;
ip.SourceAddr.Prefix+=0x01;                      //填充 45 位路由前缀
ip.SourceAddr.Prefix<<=16;
ip.SourceAddr.Prefix+=0x01;                      //填充 16 位子网号
ip.SourceAddr.Prefix=hton64(ip.SourceAddr.Prefix);
ip.SourceAddr.MacAddr[0]=char(0x00);             //填充 8 字节 MAC 地址
ip.SourceAddr.MacAddr[1]=char(0x00);
ip.SourceAddr.MacAddr[2]=char(0x80);
ip.SourceAddr.MacAddr[3]=char(0xFF);
ip.SourceAddr.MacAddr[4]=char(0xFE);
ip.SourceAddr.MacAddr[5]=char(0x18);
ip.SourceAddr.MacAddr[6]=char(0x6E);
ip.SourceAddr.MacAddr[7]=char(0xE5);

//填充目的 IP 地址
ip.DestinAddr.Prefix=0x1;                        //填充 3 位地址前缀
ip.DestinAddr.Prefix<<=45;
ip.DestinAddr.Prefix=+0x02;                      //填充 45 位路由前缀
ip.DestinAddr.Prefix<<=16;
ip.DestinAddr.Prefix=+0x02;                      //填充 16 位子网号
ip.DestinAddr.Prefix=hton64(ip.DestinAddr.Prefix);
ip.DestinAddr.MacAddr[0]=char(0x00);             //填充 8 字节 MAC 地址
ip.DestinAddr.MacAddr[1]=char(0x00);
ip.DestinAddr.MacAddr[2]=char(0xE4);
ip.DestinAddr.MacAddr[3]=char(0xFF);
ip.DestinAddr.MacAddr[4]=char(0xFE);
ip.DestinAddr.MacAddr[5]=char(0x86);
ip.DestinAddr.MacAddr[6]=char(0x3A);
ip.DestinAddr.MacAddr[7]=char(0xDC);

//填充 TCP 伪头部各字段
```

```
psd_head psd={0};
memcpy(psd.SourceAddr,&(ip.SourceAddr),16);
memcpy(psd.DestinAddr,&(ip.DestinAddr),16);
psd.Reserved=0;
psd.Protocol=ip.NextHead;
psd.TcpLen=sizeof(tcp_head)+sizeof(tcp_data);

//填充 TCP 头部各字段
tcp_head tcp={0};
tcp.SourcePort=1000;
tcp.DestinPort=1000;
tcp.Sequence=0;
tcp.Acknowledge=0;
tcp.HeadLen=(sizeof(tcp_head)/sizeof(unsigned int)<<4|0);
tcp.WindowsLen=htons((unsigned short)10000);
tcp.TcpChecksum=0;
tcp.UrgePoint=0;

//计算 TCP 包(包括伪头部与数据)校验和
memset(check,0,65535);
memcpy(check,&psd,sizeof(psd_head));
memcpy(check+sizeof(psd_head),&tcp,sizeof(tcp_head));
memcpy(check+sizeof(psd_head)+sizeof(tcp_head),tcp_data,sizeof(tcp_data));
tcp.TcpChecksum=checksum(check,sizeof(psd_head)+sizeof(tcp_head)+
sizeof(tcp_data));

//依次写入 IP 头部、TCP 头部与数据
outfile.write((char*)&ip,sizeof(ip_head));
outfile.write((char*)&tcp,sizeof(tcp_head));
outfile.write(tcp_data,sizeof(tcp_data));
outfile.seekg(8,ios::beg);

//显示 IP 头部的部分字段与数据部分
cout<<endl<<"IPv6 头部与数据字段";
cout<<endl<<"版本:"<<(ntohl(ip.Version)>>28);
cout<<endl<<"有效载荷长度:"<<ntohs(ip.PayloadLen);
cout<<endl<<"下一个头部:"<<(int)ip.NextHead;
cout<<endl<<"源 IP 地址:";
for(int i=0;i<16;i++)                    //输出源 IP 地址
{
    if(i==15)
        cout<<hex<<outfile.get();
    else
        cout<<hex<<outfile.get()<<":";
}
```

```
        cout<<endl<<"目的 IP 地址:";
        for(int i=0;i<16;i++)                          //输出目的 IP 地址
        {
            if(i==15)
                cout<<hex<<outfile.get();
            else
                cout<<hex<<outfile.get()<<":";
        }
        cout<<endl<<"数据字段:"<<tcp_data<<endl;

        cout<<endl<<"IPv6 包封装完成"<<endl;
        outfile.close();                               //关闭输出文件
    }
```

图8-6给出了IPv6数据包的封装过程。程序命令行输入为Ipv6Encap output。程序构造IPv6头部、TCP头部与伪头部的数据结构,然后依次填充IPv6头部、TCP头部的字段与数据部分,最后将填充好的IPv6数据包内容写入output。

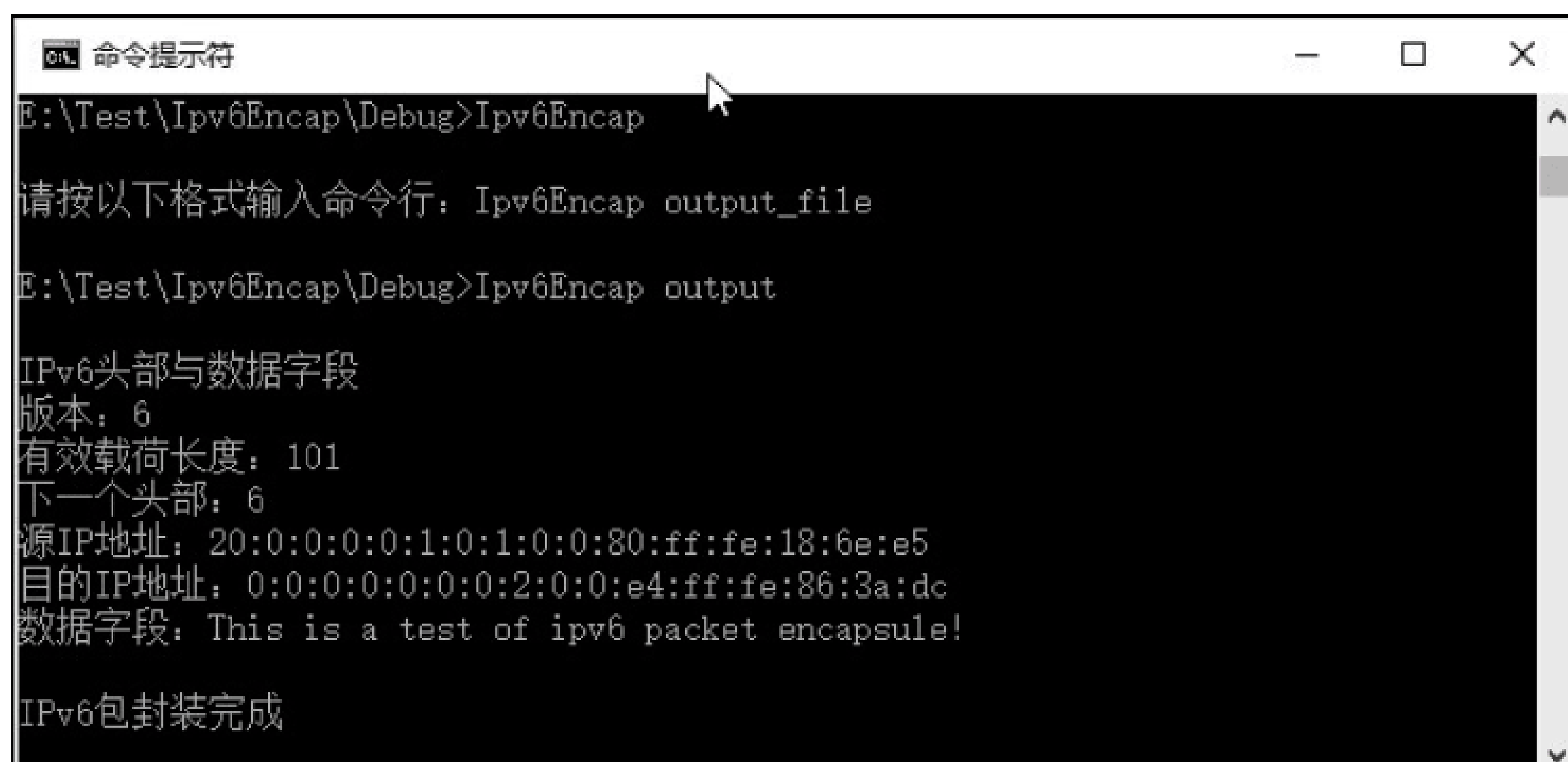


图8-6　IPv6数据包的封装过程

8.4　练　习　题

根据协议规定的IPv6数据包的标准格式,编写程序解析IPv6包结构(包括IPv6头部与TCP头部),然后将解析后的IPv6包内容显示出来。在本练习中为了简便起见,IPv6包内容可以从输入文件中获得,并且不需要验证TCP头部与数据部分的校验和。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为Ipv6Parse.exe,则程序的命令行格式为:

```
Ipv6Parse input_file
```

其中,input_file为输入文件。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
IP头部与数据字段
版本:xx
优先级:xx
流标号:xx
有效载荷长度:xx
下一个头部:xx
跳步限制:xx
源 IP地址:xx:xx:xx:xx:xx:xx:xx:xx
目的 IP地址:xx:xx:xx:xx:xx:xx:xx:xx
数据字段:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第9章 发现网络中的活动主机

9.1 设计目的

ICMP 协议是 TCP/IP 协议族中的重要部分。IP 协议的最大优点是简洁,但是它缺少差错控制与查询机制。设计 ICMP 协议的目的是补充 IP 的功能。本章练习的目的是根据 ICMP 协议的基本原理,通过封装、发送、接收与解析 ICMP 数据包,了解 ICMP 包结构中各个字段的用途,从而深入理解与认识 ICMP 协议的作用。

9.2 相关知识

本章涉及的相关知识包括 ICMP 协议的概念与 ICMP 数据包结构。

9.2.1 ICMP 协议的基本概念

IP 协议提供的是一种无连接的、尽力而为的服务。在 IP 数据包通过网络传输的过程中,出现各种传输错误是不可避免的。例如,IP 数据包因超过生存时间而被丢弃,目的主机在预定时间内无法收到所有分片。这些错误都可能造成数据传输失败,而源节点无法知道 IP 数据包是否到达目的节点,也无法知道在传输过程中出现哪种错误。也就是说,IP 协议的缺点是缺少差错控制与查询机制。互联网控制报文协议(Internet Control Message Protocol,ICMP)就是为解决这个问题而设计的。

ICMP 协议本身是一个网络层的协议。但是,ICMP 数据包并不是直接传送给下面的数据链路层,而是封装成 IP 数据包后传送给数据链路层。如果只从这点来看,ICMP 协议的层次应该高于 IP 协议。但是,ICMP 功能是解决 IP 协议可能出现的差错问题,它不能独立于 IP 协议而单独存在,因此还是应该将它看作是 IP 协议的一部分。图 9-1 给出了 ICMP 数据包与 IP 数据包的关系。ICMP 头部与 ICMP 数据都是作为 IP 数据来封装的。IP 包头部中的协议字段值为 1,则说明这个 IP 数据包是一个 ICMP 数据包。

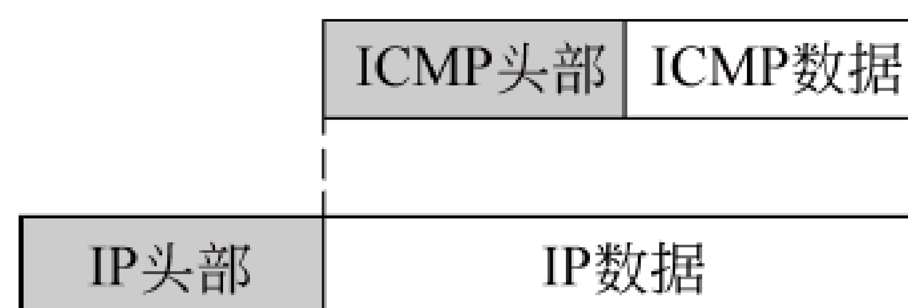


图 9-1 ICMP 数据包与 IP 数据包的关系

在当前基于 IPv4 的网络层协议体系中,实现差错通知功能的是 ICMP 协议的第 4 版,

简称 ICMPv4。随着下一代 IP 协议 IPv6 的不断完善及其应用，在基于 IPv6 的网络层协议体系中，ICMP 协议版本也会相应过渡到 ICMPv6。该协议的主要变化表现在两个方面：一方面是去掉过时的报文类型，定义一些新的报文类型；另一方面是合并原来的 IGMP、ARP 等协议的功能。

9.2.2 ICMP 数据包的类型

ICMP 数据包类型可以分为两类：差错通知报文与查询报文。表 9-1 给出了 ICMP 数据包的具体类型。其中，ICMP 差错通知报文主要分为 5 种：目的不可达报文、源主机抑制报文、超时报文、参数问题报文与重定向报文。IP 协议提供无连接的分组传输服务，在协议中并没有设计流量控制功能，源主机、路由器与目的主机之间没有协调机制。由于路由器的缓冲区长度有限，如果路由器接收分组速度比转发速度慢，这时就会因缓冲区溢出而丢弃某些分组。“源抑制”是指路由器或主机因拥塞而丢弃分组时，向源主机发送源抑制报文。超时报文用于解决 IP 分组在网络中无限转发问题。重定向报文用于解决主机与路由器的路由表差异问题。

表 9-1　ICMP 数据包的主要类型

类　型	代码	功能描述
0	0	回送应答(Ping 应答)
3(目的不可达报文)	0	网络不可达
	1	主机不可达
	2	协议不可达
	3	端口不可达
	4	需要分片，但标记为不可分片
	5	源站选路失败
	6	目的网络不可知
	7	目的主机不可知
4(源主机抑制报文)	0	源主机抑制(数据流控制)
5(重定向报文)	0	网络重定向
	1	主机重定向
	2	服务类型和网络重定向
	3	服务类型和主机重定向
8	0	回送请求(Ping 请求)
9	0	路由器通告
10	0	路由器查询
11(超时报文)	0	传输期间生存期减为 0
	1	数据包组装期间生存期减为 0

续表

类 型	代码	功 能 描 述
12(参数问题报文)	0	各种 IP 头部错误
	1	缺少必要的选项
13	0	时间戳请求
14	0	时间戳应答
17	0	地址掩码请求
18	0	地址掩码应答

ICMP 目的不可达是常用的 ICMP 差错通知报文。当路由器因无法向目的主机交付而丢弃 IP 分组时,路由器或目的主机向源主机发送 ICMP 目的不可达报文。最初,目的不可达报文主要有 5 种:网络不可达、主机不可达、协议不可达、端口不可达和源路由失败。后来,目的不可达报文增加了几种:网络不可知、主机不可知、网络被禁用、主机被禁用和防火墙过滤等。这里,网络不可达与网络不可知的区别是:网络不可达是指路由器知道目的网络存在,但是无法将 IP 分组交付网络;网络不可知是指路由器不知道目的网络存在。主机不可达与主机不可知的区别和网络不可达与网络不可知类似。

ICMP 查询报文的设计目标是解决网络故障的诊断问题。ICMP 差错控制报文是单向、单个出现的,而 ICMP 查询报文是双向、成对出现的。ICMP 查询报文主要分为 4 种:回送请求与应答、时间戳请求与应答、地址掩码请求与应答、路由器查询与通告。其中,回送请求用于主机检查某台主机获得路由器是否可达。时间戳请求提供了一个简单的时钟同步协议,可用于获得 IP 分组在两台主机之间往返传输所需要的时间。地址掩码请求用于主机获得所在网络的子网掩码。路由器查询用于主机查询所在网络的本地路由器地址,路由器通告用于路由器向外广播自己的路由信息。

9.2.3 ICMP 数据包的结构

ICMP 协议的设计初衷是报告 IP 协议执行中的错误,由路由器或目的主机向源主机报告传输出错的原因,真正的差错处理功能需要由高层协议来完成。RFC777 是最早出现的 ICMP 协议文档,它描述了 ICMP 协议的基本内容。RFC792 文档对 ICMP 报文类型加以修改与补充。RFC1256 文档增加了路由器查询与通告报文。图 9-2 给出了 ICMP 数据包的结构。ICMP 数据报分为两部分:ICMP 头部与 ICMP 数据。ICMP 头部的长度为 4B。ICMP 数据部分的长度是可变的,具体内容由 ICMP 包的类型决定。

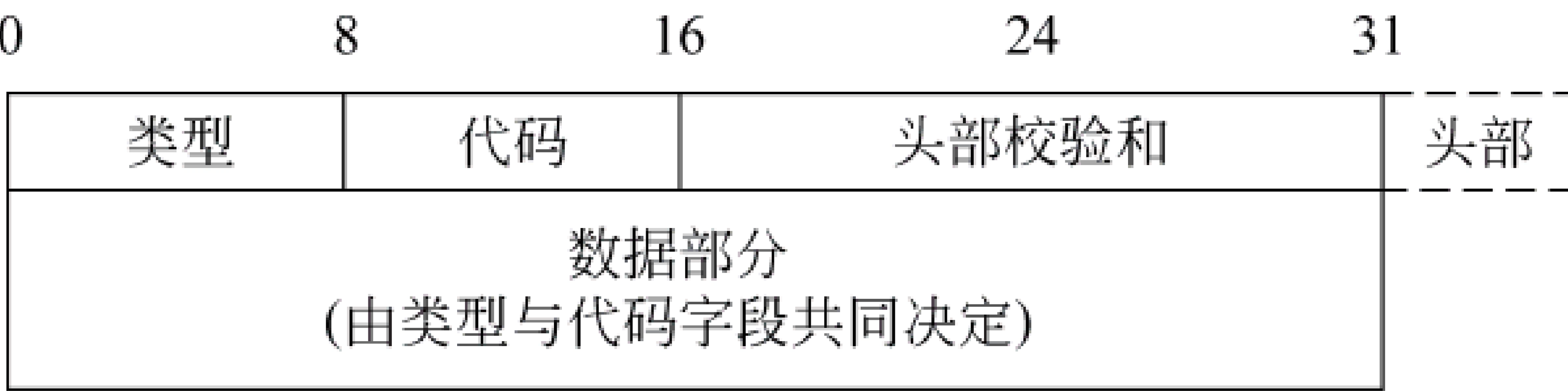


图 9-2 ICMP 数据包的结构

ICMP 头部由以下字段组成。

1. 类型

类型(type)字段的长度为8位,表示ICMP报文的基本类型。目前,类型字段主要有13个数值,分别表示13类的ICMP报文。例如,3表示目的不可达报文,5表示重定向报文,8表示回送请求报文,10表示路由器查询报文。有些类型的ICMP报文已在近年被废除,例如,15表示的信息请求报文、16表示的信息应答等。实际上,由类型与代码字段共同标识具体的ICMP类型。

2. 代码

代码(code)字段的长度为8位,表示ICMP报文的子类型。对于ICMP差错通知类报文,每种报文又可以分为多种子类型。目的不可达报文又可细分为8种子类型,例如,0表示的网络不可达、1表示的主机不可达等。重定向报文又可细分为4种子类型,例如,0表示的网络重定向、1表示的主机重定向等。对于ICMP查询类报文,例如回送请求与应答、路由器查询与通告,它们的代码字段都只有0这个值。

3. 头部校验和

头部校验和(head checksum)字段的长度为16位,用来检查ICMP包头部在传输中是否出错,其计算方法为IP头部校验和的计算方法相同。头部校验和字段的校验范围为ICMP头部与ICMP数据。

9.2.4 ICMP回送请求与应答

本章习题的目的是判断网络中的主机状态,使用的是ICMP类型中的回送请求与应答。图9-3给出了ICMP回送报文的结构。这里,类型字段的值为8,表示ICMP回送请求;类型字段的值为0,表示ICMP回送应答。代码字段的值都是0。ICMP回送报文有两个特殊字段:标识符(identifier)的长度为16位,表示回送请求与应答的对应关系;序列号(sequence number)字段的长度为16位,表示回送请求与应答的编号。由于回送请求与应答都是成对出现的,因此标识符与序列号应分别填充相同的值。



图9-3 ICMP回送报文的结构

源节点向目的节点发送ICMP回送请求后,等待接收目的节点返回ICMP回送应答。如果源节点在规定时间内收到应答信息,目的节点处于活动状态;否则,目的节点处于关闭或不应答状态。实际上,网络用户日常使用的Ping命令就是用于发现网络中的活动主机。图9-4给出了Windows系统中的Ping命令。这里,源主机(本机)向目的主机(192.168.1.1)发送4个回送请求,目的主机向源主机返回4个回送应答,每个报文中包括三个参数:报文长度(bytes)、响应时间(time)与生存时间(TTL)。

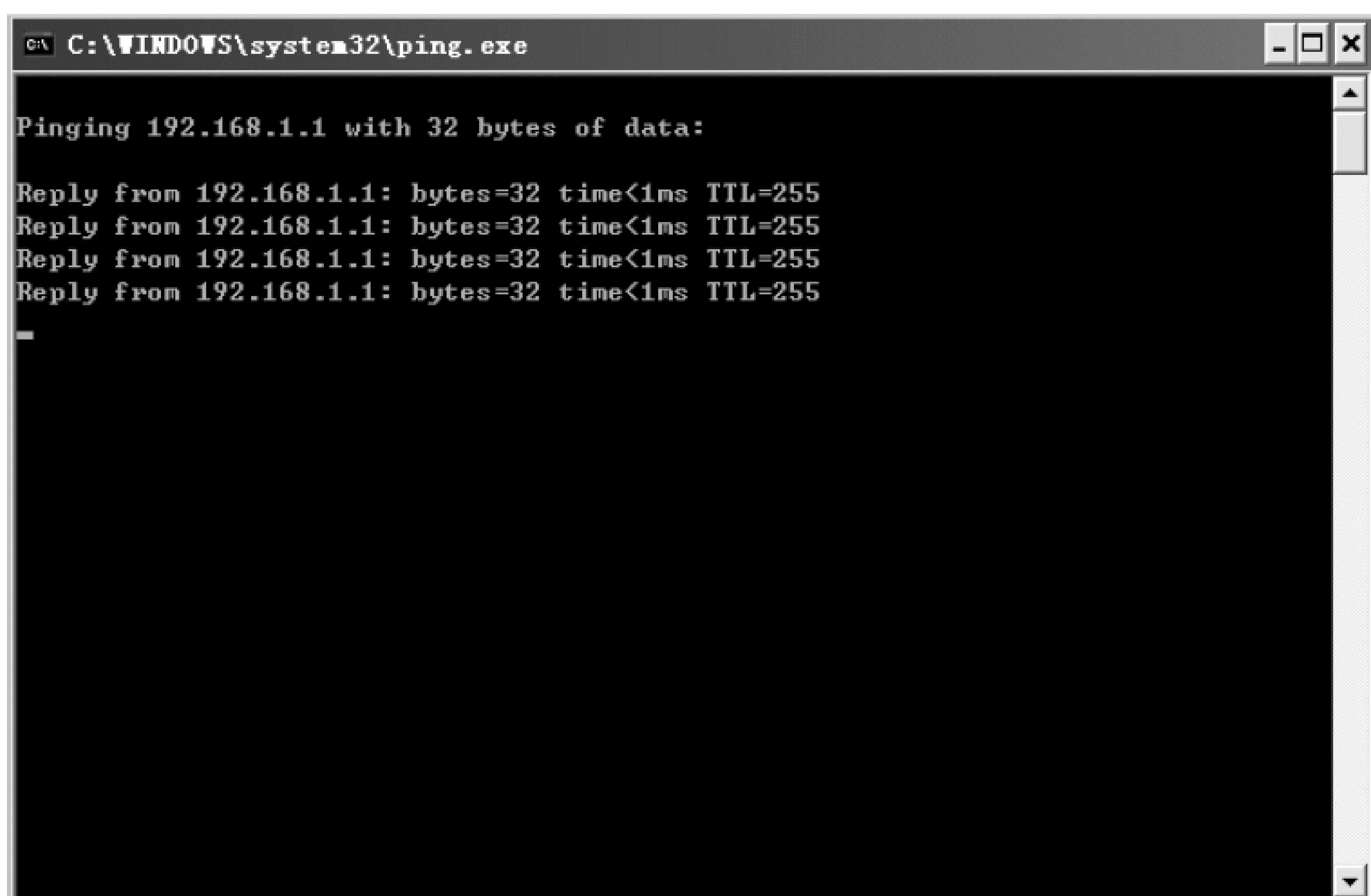


图 9-4　Windows 系统中的 Ping 命令

9.3　例题分析

9.3.1　设计要求

根据协议规定的 ICMP 数据包的标准格式,编写程序向目的主机发送 ICMP 回送请求,并对目的主机返回的 ICMP 回送应答进行解析,以判断目的主机是否处于活动状态。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 ScanHost.exe,则程序的命令行格式为:

```
ScanHost host_addr
```

其中,host_addr 为目的主机的 IP 地址。

(2) 要求将目的主机状态显示在控制台上,具体格式为:

```
开始主机扫描
目的主机+IP 地址:活动状态 (或关闭状态)
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

9.3.2　关键问题

1. 创建原始套接字

为了实现发送与接收 ICMP 数据包,首先需要调用 socket()函数创建原始套接字,其中

的 SOCK_RAW 表示创建原始套接字，IPPROTO_ICMP 表示采用 ICMP 协议。接着，需要调用 setsockopt 函数设置发送与接收超时，SO_SNDTIMEO 表示发送超时，SO_RCVTIMEO 表示接收超时，超时时间均设置为 1000ms。如果源节点在接收超时内没有收到 ICMP 应答，则说明目的节点没有处于活动状态。

下面给出创建原始套接字的伪代码：

```
//套接字异步启动
WSAStartup(MAKEWORD(2,2),&WSAData);
//创建原始 Socket
sock=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
//设置发送超时
int send_timeout=1000;
setsockopt(sock,SOL_SOCKET,SO_SNDTIMEO,&send_timeout,sizeof(send_timeout));
//设置接收超时
int recv_timeout=1000;
setsockopt(sock,SOL_SOCKET,SO_RCVTIMEO,&recv_timeout,sizeof(recv_timeout));
```

2. 定义 ICMP 头部的数据结构

ICMP 头部与 ICMP 数据要作为 IP 数据，与 IP 头部封装成 IP 数据包才能够发送。在对 ICMP 头部各字段进行填充之前，首先需要构造 ICMP 头部的数据结构，该数据结构与图 9-2 的 ICMP 头部结构一致。另外，还需要构造 IP 头部的数据结构，这部分已经在第 7 章介绍过。

下面给出构造 ICMP 头部的伪代码：

```
//定义 ICMP 头部结构
typedef struct ICMP_HEAD
{
    unsigned char Type;
    unsigned char Code;
    unsigned short HeadChecksum;
    unsigned short Identifior;
    unsigned short Sequence;
}icmp_head;
```

3. 填充与发送 ICMP 包

在填充 ICMP 数据包的过程中，需要分别填充 IP 头部、ICMP 头部与 ICMP 数据。由于 ICMP 回送请求的类型为 8、代码为 0，因此需要将它们填入 ICMP 头部的相应字段。ICMP 头部的校验和需要进行计算，首先为校验和字段赋初值 0，然后将校验和计算函数 checksum()的结果填入校验和字段。

下面给出填充与发送 ICMP 包的伪代码：

```
//填充 ICMP 数据包
char icmp_data[MAX_PACKET];
icmp_head * icmp_hdr;
```

```
int icmpsize;
memset(icmp_data,0,MAX_PACKET);
icmpsize=DEF_PACKET+sizeof(icmp_head);
icmp_hdr=(icmp_head*)icmp_data;
icmp_hdr->Type=ICMP_ECHO;
icmp_hdr->Identifior=(unsigned short)GetCurrentThreadId();
icmp_hdr->HeadChecksum=0;
icmp_hdr->HeadChecksum=checksum((unsigned short*)icmp_data,icmpsize);
//初始化目的地址
sockaddr_in dest;
memset(&dest,0,sizeof(dest));
dest.sin_family=AF_INET;
dest.sin_addr.s_addr=inet_addr(argv[1]);
//发送 ICMP 数据包
sendto(sock,icmp_data,icmpsize,0,(struct sockaddr*)&dest, sizeof(dest));
```

4. 接收与解析 ICMP 包

如果目的主机处于活动状态,它会向源主机发送一个 ICMP 回送应答。源主机接收到 ICMP 包后需要对它进行解析,根据 IP 头部中的地址字段可以获得 IP 地址,根据 ICMP 头部的类型字段可以判断是否为回送应答(类型为 0)。

下面给出接收与解析 ICMP 包的伪代码:

```
//初始化源地址
sockaddr_in from;
int fromlen=sizeof(from);
memset(&from,0,sizeof(from));
char* recvbuf=new char[MAX_PACKET+sizeof(ip_head)];
//接收 ICMP 数据包
int nRecv=recvfrom(sock,recvbuf,MAX_PACKET+sizeof(ip_head),0,(struct sockaddr*),&from,
&fromlen);
//解析 ICMP 数据包
ip_head *iphdr;
icmp_head *icmphdr;
unsigned short ip_size;
iphdr=(ip_head*)recvbuf;
ip_size=(iphdr->HeadLen&0x0f)*4;
icmphdr=(icmp_head*)(recvbuf+ip_size);
//判断 ICMP 报文类型
if(nRecv<ip_size+ICMP_MIN)
    …
if(icmphdr->Type!=ICMP_ECHO_REPLY)
    …
if(icmphdr->Identifior!=(unsigned short)GetCurrentThreadId())
    …
```

5. 程序流程图

图 9-5 给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要有一个主机 IP 地址。如果命令行参数的个数不是一个,则程序在输出错误信

息后退出。在主程序的流程中,需要判断是否到达接收超时,以及是否包含回送应答。

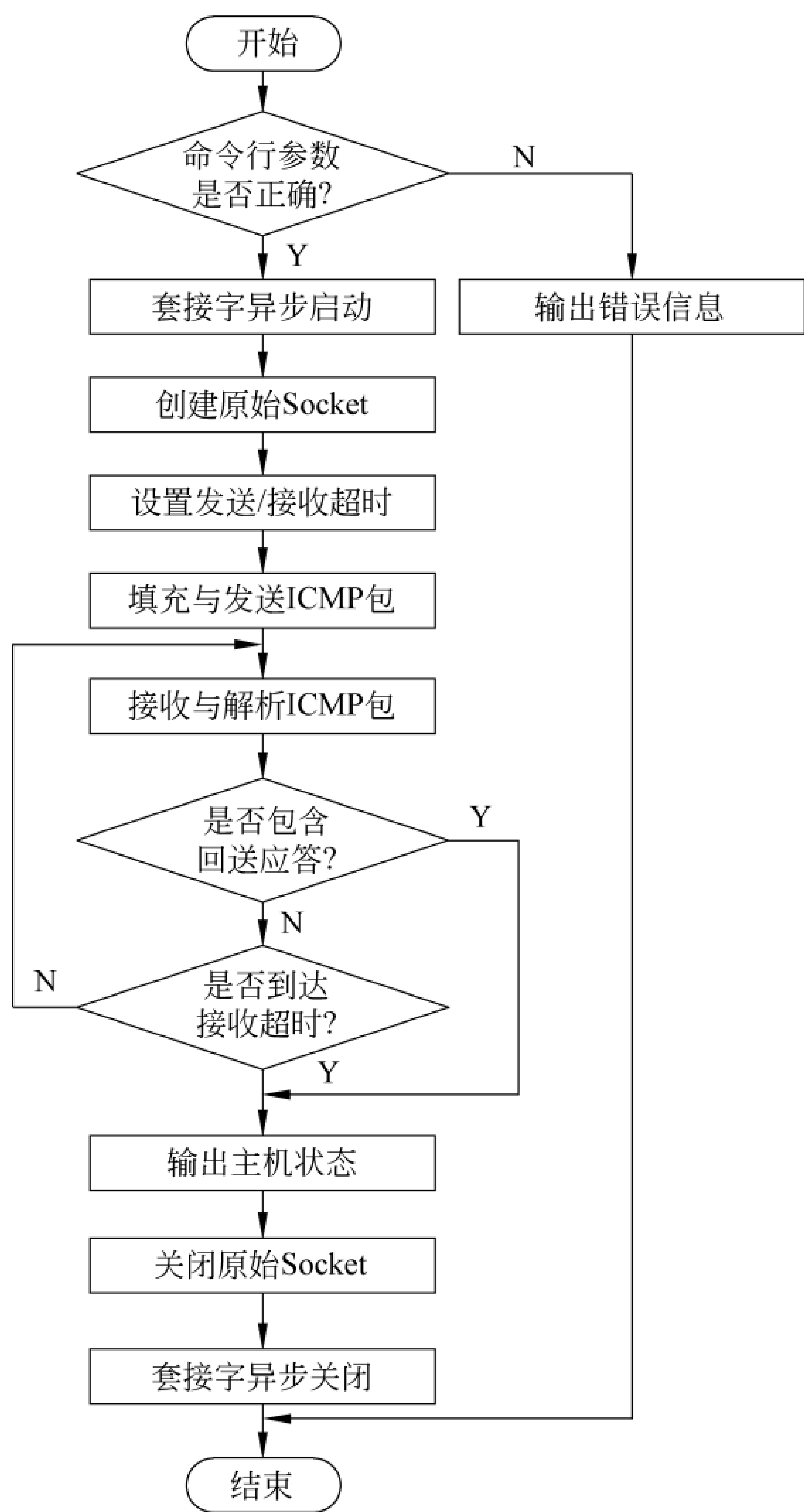


图 9-5　主程序流程图

9.3.3　程序源代码

下面给出主机扫描程序的源代码:

```
//ScanHost.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "time.h"
#include "winsock2.h"
#include "iostream"
using namespace std;
```

```
#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib

typedef struct IP_HEAD                           //定义 IP 头部结构
{
    union
    {
        unsigned char Version;                   //版本(字节前 4 位)
        unsigned char HeadLen;                   //头部长度(字节后 4 位)
    };
    unsigned char ServiceType;                   //服务类型
    unsigned short TotalLen;                     //总长度
    unsigned short Identifier;                   //标识符
    union
    {
        unsigned short Flags;                    //标志位(字前 3 位)
        unsigned short FragOffset;               //片偏移(字后 13 位)
    };
    unsigned char TimeToLive;                    //生存周期
    unsigned char Protocol;                      //协议
    unsigned short HeadChecksum;                 //头部校验和
    unsigned int SourceAddr;                     //源 IP 地址
    unsigned int DestinAddr;                     //目的 IP 地址
}ip_head;

typedef struct ICMP_HEAD                         //定义 ICMP 头部结构
{
    unsigned char Type;                          //ICMP 类型(回送请求为 8)
    unsigned char Code;                          //ICMP 子类型
    unsigned short HeadChecksum;                 //头部校验和
    unsigned short Identifior;                   //ICMP 包 ID 号
    unsigned short Sequence;                     //ICMP 包序列号
}icmp_head;
#define ICMP_ECHO 8                              //请求回送
#define ICMP_ECHO_REPLY 0                        //请求回应
#define ICMP_MIN 8                               //最小 ICMP 包长度
#define DEF_PACKET 32                            //默认数据包长度
#define MAX_PACKET 1024                          //数据最大长度

unsigned short checksum(unsigned short * buffer,int size)
{                                                //校验和计算函数
    unsigned long cksum=0;
    while(size>1)
    {
```

```
        cksum+=*buffer++;
        size-=sizeof(unsigned short);
    }
    if(size)
        cksum+=*(unsigned char*)buffer;
    cksum=(cksum>>16)+(cksum&0xffff);
    cksum+=(cksum>>16);
    return(unsigned short)(~cksum);
}

void main(int argc,char * argv[])
{
    if(argc!=2)                                     //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:ScanHost host_addr"<<endl;
        return;
    }

    WSADATA WSAData;
    if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)       //套接字异步启动
    {
        cout<<endl<<"WSAStartup初始化失败!"<<endl;
        return;
    }

    SOCKET sock=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
    if(sock==INVALID_SOCKET)                        //创建原始Socket
    {
        cout<<endl<<"创建Socket失败!"<<endl;
        return;
    }

    int send_timeout=1000;                          //设置发送超时
    if(setsockopt(sock,SOL_SOCKET,SO_SNDTIMEO,(char*)&send_timeout,
    sizeof(send_timeout))==SOCKET_ERROR)
    {
        cout<<endl<<"设置发送超时失败!"<<endl;
        return;
    }

    int recv_timeout=1000;                          //设置接收超时
    if(setsockopt(sock,SOL_SOCKET,SO_RCVTIMEO,(char*)&recv_timeout,
    sizeof(recv_timeout))==SOCKET_ERROR)
    {
```

```
        cout<<endl<<"设置接收超时失败!"<<endl;
        return;
    }

    sockaddr_in dest;                              //初始化目的地址
    memset(&dest,0,sizeof(dest));
    dest.sin_family=AF_INET;
    dest.sin_addr.s_addr=inet_addr(argv[1]);       //填入扫描 IP 地址
    cout<<endl<<"开始主机扫描";

    //填充 ICMP 数据包
    char icmp_data[MAX_PACKET];
    icmp_head * icmp_hdr;
    int icmpsize;
    memset(icmp_data,0,MAX_PACKET);
    icmpsize=DEF_PACKET+sizeof(icmp_head);         //计算 ICMP 包长度
    icmp_hdr=(icmp_head*)icmp_data;
    icmp_hdr->Type=ICMP_ECHO;                      //设置类型信息
    icmp_hdr->Identifior=(unsigned short)GetCurrentThreadId();
                                                   //设置 ID 为当前线程
    icmp_hdr->HeadChecksum=0;
    icmp_hdr->HeadChecksum=checksum((unsigned short*)icmp_data,icmpsize);
                                                   //计算 ICMP 包校验和

    //发送 ICMP 数据包
    int nSend=sendto(sock,icmp_data,icmpsize,0,(struct sockaddr*)&dest,
sizeof(dest));
    if(nSend==SOCKET_ERROR||nSend<icmpsize)
    {
        cout<<endl<<"ICMP 包发送失败!"<<endl;
        return;
    }

    sockaddr_in from;                              //初始化源地址
    int fromlen=sizeof(from);
    memset(&from,0,sizeof(from));
    char* recvbuf=new char[MAX_PACKET+sizeof(ip_head)];

    //接收 ICMP 数据包
    int nRecv=recvfrom(sock,recvbuf,MAX_PACKET+sizeof(ip_head),0,(struct sockaddr*)
    &from,&fromlen);
    if(nRecv==SOCKET_ERROR)
```

```
        {
            cout<<endl<<"主机"<<inet_ntoa(dest.sin_addr)<<":关闭状态"<<endl;
            return;
        }

        ip_head * iphdr;
        icmp_head * icmphdr;
        unsigned short ip_size;
        iphdr=(ip_head*)recvbuf;
        ip_size=(iphdr->HeadLen&0x0f) * 4;
        icmphdr=(icmp_head*)(recvbuf+ip_size);        //跳过 IP 包头部

        //对接收 ICMP 包进行判断
        bool icmptype=true;
        if(nRecv<ip_size+ICMP_MIN)
        {
            cout<<endl<<"接收包太短,丢弃!"<<endl;
            icmptype=false;
        }
        if(icmphdr->Type!=ICMP_ECHO_REPLY)
        {
            cout<<endl<<"不是回送响应,丢弃!"<<endl;
            icmptype=false;
        }
        if(icmphdr->Identifior!=(unsigned short)GetCurrentThreadId())
        {
            cout<<endl<<"ID号不相符丢弃!"<<endl;
            icmptype=false;
        }
        if(icmptype==true)
            cout<<endl<<"主机"<<inet_ntoa(dest.sin_addr)<<":活动状态"<<endl;

        closesocket(sock);                              //关闭原始 Socket
        WSACleanup();                                   //套接字异步关闭
    }
```

图 9-6 给出了网络中主机的扫描过程。程序命令行的输入依次为 ScanHost 10.134.37.128 与 ScanHost 10.134.37.127。程序依次向主机 10.134.37.128 与主机 10.134.37.127 发送 ICMP 回送请求,根据是否接收到 ICMP 回送应答,判断出主机 10.134.37.128 与主机 10.134.37.127 分别处于活动状态与关闭状态。

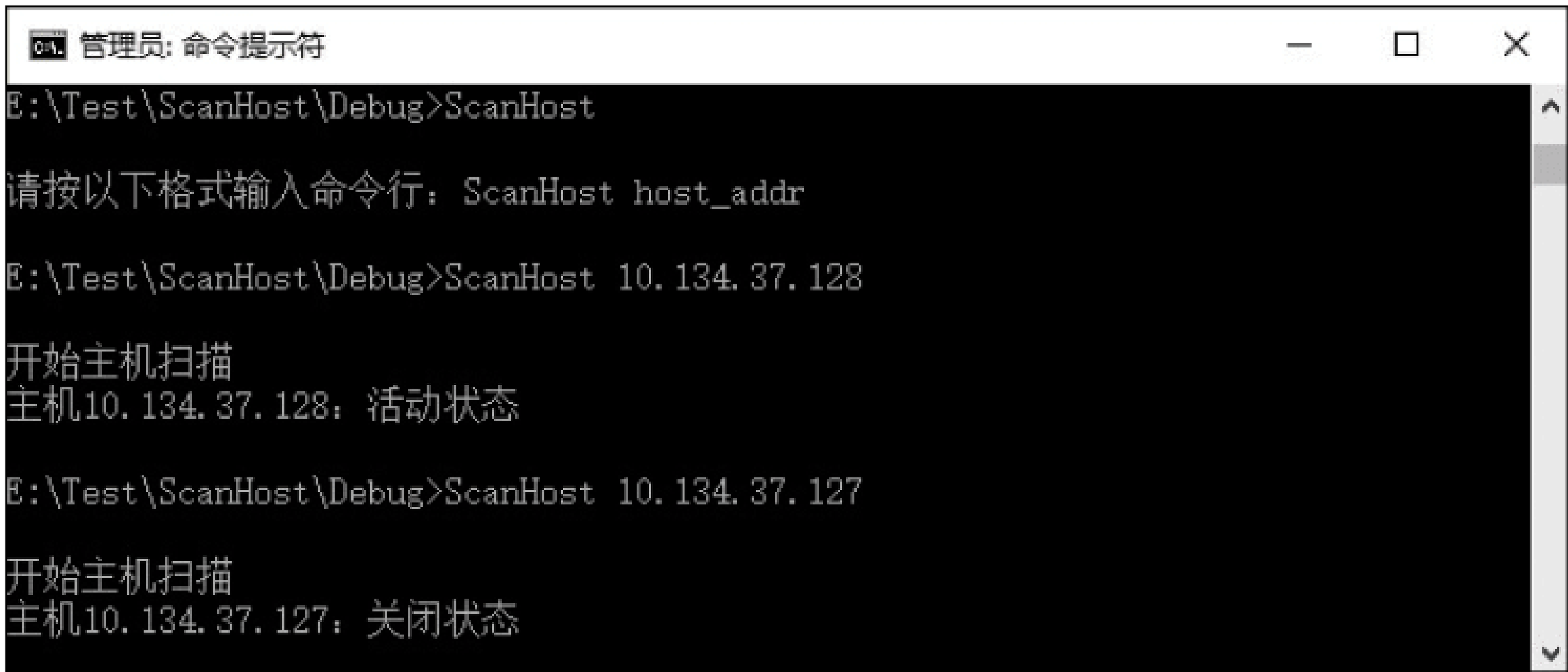


图 9-6　网络中主机的扫描过程

9.4　练　习　题

根据协议规定的 ICMP 数据包的标准格式，编写程序向指定子网中的目的主机(例如从192.168.1.1 到 192.168.1.11)发送 ICMP 包，并对目的主机返回的 ICMP 包进行解析，以发现处于活动状态的主机。在本练习中只显示活动主机的 IP 地址，并采用多线程来提高主机扫描速度。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 ScanHost.exe，则程序的命令行格式为：

```
ScanHost start_addr end_addr
```

其中，start_addr 为开始搜索的 IP 地址，end_addr 为结束搜索的 IP 地址。

(2) 要求将目的主机状态显示在控制台上，具体格式为：

```
开始主机扫描
活动主机:xx.xx.xx.xx
活动主机:xx.xx.xx.xx
...
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 10 章 发现服务器开启的 TCP 端口

10.1 设 计 目 的

网络服务是以客户机/服务器模式工作，服务器在某些特定端口上提供网络服务，等待客户机发出的服务请求。传输层提供的传输服务包括 TCP 与 UDP 两种类型。本章练习的目的是，通过发现服务器开启的 TCP 端口，了解传输层的基本功能与协议类型，掌握网络服务、端口的概念与相互关系。

10.2 相 关 知 识

本章涉及的相关知识包括传输层的概念与端口号的分配方法。

10.2.1 传输层的基本概念

从网络分层结构的角度来看，网络层及以下各层实现网络主机之间的数据通信，但是数据通信并不是组建计算机网络的最终目的。计算机网络的本质是实现分布在不同地理位置的主机之间的资源共享，以便实现在应用层提供的各种网络服务。传输层是 OSI 参考模型中的重要层次，主要作用是实现网络环境中的分布式进程通信，为实现应用层的各种网络服务功能提供传输服务，因此传输层是整个网络协议结构中的重要部分。传输层在网络体系结构中起到承上启下的作用。

传输层协议利用网络层协议所提供的服务，在源主机与目的主机的应用进程之间实现“端到端”服务，也就是两台主机之间的分布式进程通信。图 10-1 给出了传输层与网络层协议的关系。在两台主机的应用进程之间进行通信，需要穿过一个结构相当复杂的通信子网，它实际上是由路由器、通信线路与其他网络设备构成的传输网。网络层的 IP 协议为传输层提供尽力而为的分组交换服务，路由选择协议为传输层提供选择路径的路由选择服务，因此网络层协议实现的是“点到点”服务。

计算机网络是分布在不同地理位置的多台独立的计算机系统的集合，联网的每台计算机的资源是由自己的操作系统来管理。用户共享的网络资源与网络所能提供的服务，最终是通过网络环境中的分布式进程通信来实现的。这种网络环境中的进程通信与单机系统内部的进程通信的主要区别在于网络主机的高度自主性。由于它们并不是在同一个主机系统

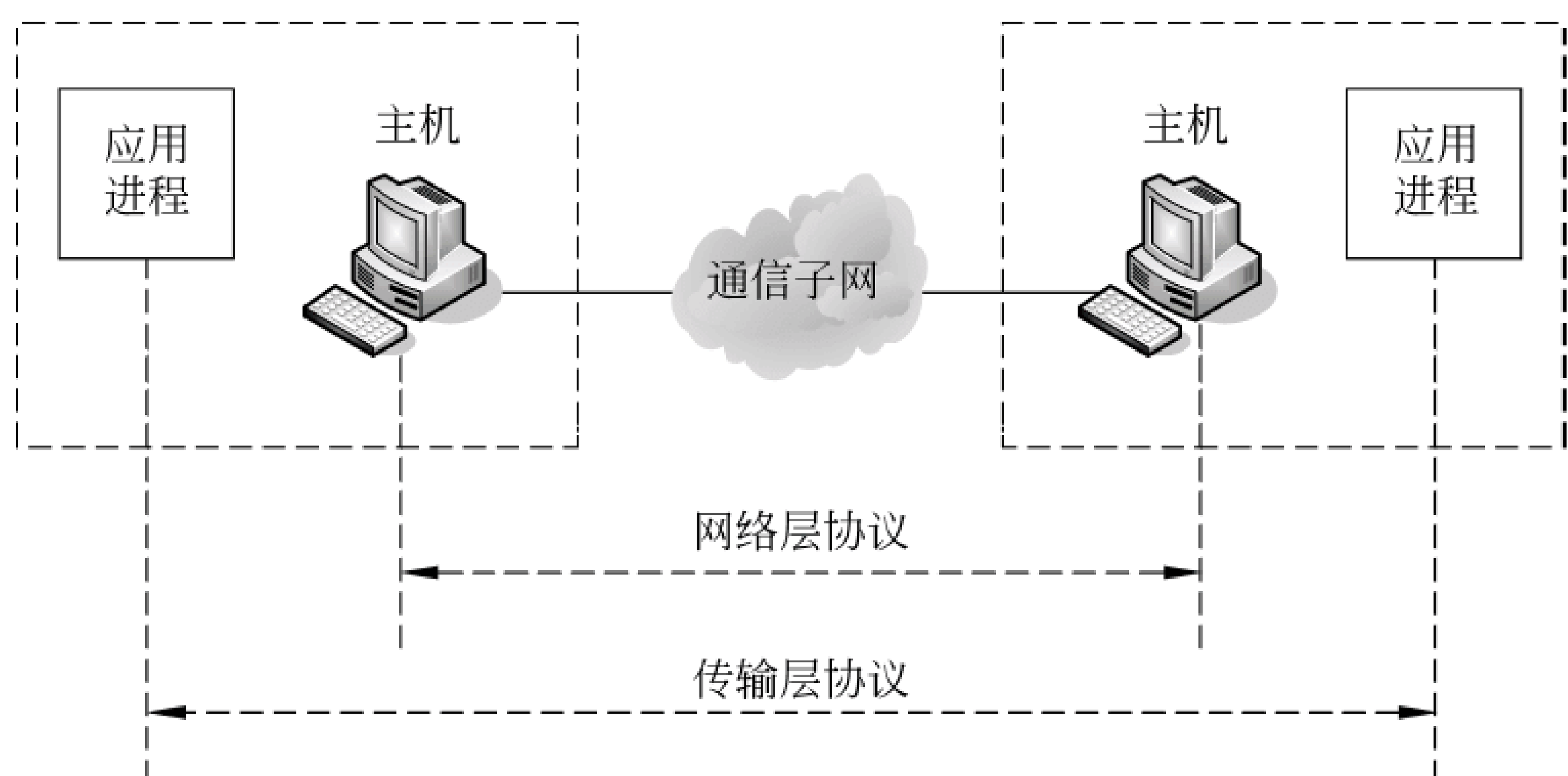


图 10-1　传输层与网络层协议的关系

中,所以没有一个统一的高层操作系统进行全局控制。分布式进程通信需要解决三个问题:①进程命名与寻址方法;②多重协议的识别;③进程间相互作用的模式。

分布式进程通信需要解决的首要问题是进程标识。同一台计算机中的不同进程可以使用进程号唯一地进行标识,只要在进程号的分配中不出现重复,那么进程标识就不会出现二义性。但是,在网络环境中不能仅使用进程号作为标识,这是由于不同主机有可能分配相同的进程号。因此,网络环境中的进程标识还需要使用主机地址。网络环境中完整的进程标识包括本地主机地址-本地进程标识、远程主机地址-远程进程标识。这个进程标识就是传输层所要解决的问题。

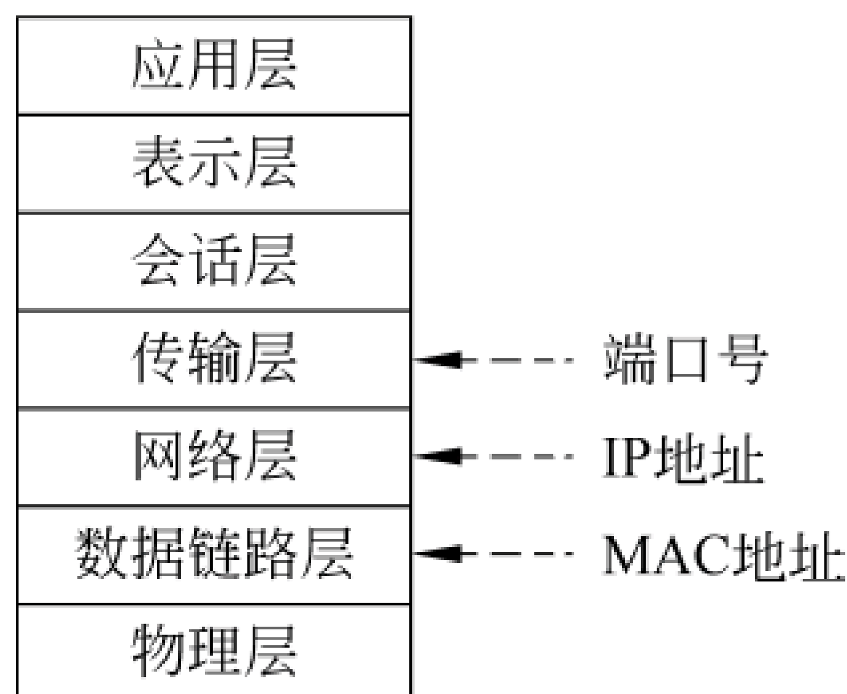


图 10-2　OSI 模型各层的编址方式

OSI 参考模型的各层都有自己的编址方式:数据链路层使用的地址是 MAC 地址,网络层使用的地址是 IP 地址,传输层使用的地址是进程地址,等等。图 10-2 给出了 OSI 模型各层的编址方式。进程地址也称为端口号(port number),端口号是应用程序对传输层协议的访问点,它是传输层协议软件的组成部分之一。传输层协议规定了一些用于服务器进程的保留端口号;用户可以申请使用未分配的非保留端口。这些保留与非保留端口号在主机中都是唯一的。因此,端口号可以作为网络环境中的主机进程标识。

10.2.2　端口号的分配

如果网络环境中的两台主机要实现进程通信,则它们必须使用相同类型的传输层协议。网络进程的唯一标识需要由三元组来表示:协议类型、IP 地址与端口号。图 10-3 给出了三元组的概念。这个协议类型是指传输层协议。传输层协议主要分为两种类型:TCP 协议与 UDP 协议。其中,TCP 协议是一种全双工的、可靠的、面向连接的传输层协议,它可以在通信双方之间提供无差错的数据传输。UDP 协议是一种全双工的、不可靠的、无连接的传输层协议。另外,针对近年出现的实时性要求高的网络应用,传输层

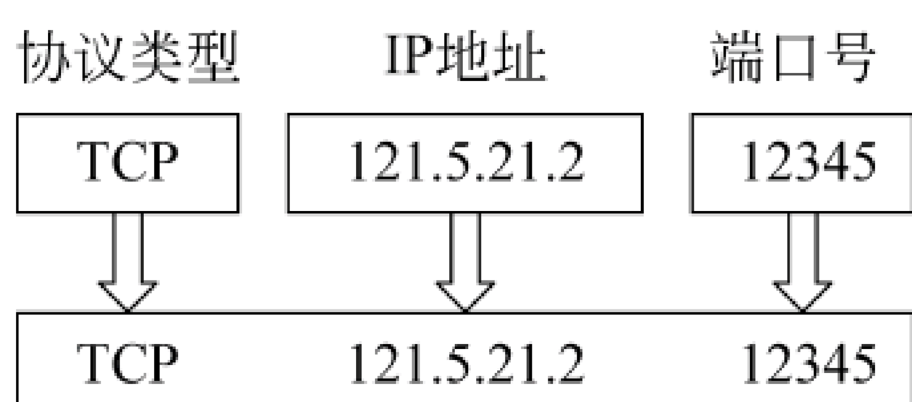


图 10-3　三元组的概念

增加了实时传输协议(RTP)与实时传输控制协议(RTCP)。

从用户应用的角度来看,端口号是应用层的网络服务对应的数字代码。端口号是一个在 0～65 535 之间的整数,TCP 与 UDP 端口号同时分配给同一种服务。端口号的分配工作由 Internet 赋号管理局(IANA)完成。端口号可以分为三种:熟知端口号、注册端口号与临时端口号。其中,熟知端口号的范围是 0～1023,它被统一分配给某种指定的网络服务;注册端口号的范围是 1024～49 151,它被分配给需要注册使用的网络服务;临时端口号的范围是 49 152～65 535,它可以被任何进程临时申请使用。

实际上,每种网络服务都采用客户机/服务器(client/server)模式。客户机与服务器是进行通信的两个应用进程。这里,客户机是使用某种网络服务的应用进程,服务器是提供某种网络服务的应用进程。客户机/服务器模式的工作过程是:客户机向服务器发送服务请求,服务器接收客户机的请求并做出响应,决定是否向客户机提供所需的数据。图 10-4 给出了不同类型端口号的作用。无论是客户机进程还是服务器进程,它们都要通过 IP 地址与端口号加以标识,这里的主要区别在于端口号的类型。

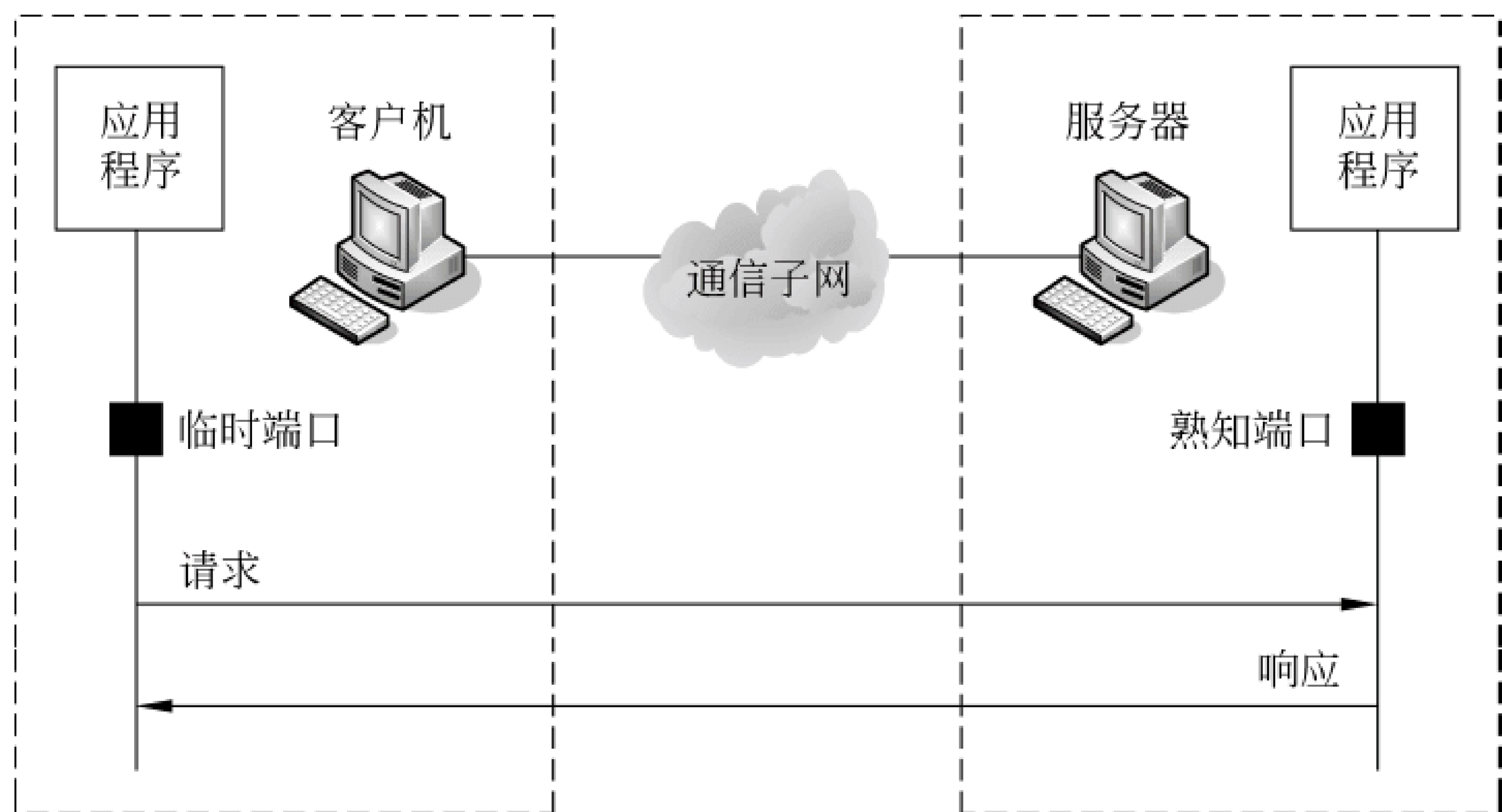


图 10-4　不同类型端口号的作用

在基于 TCP 的网络应用中,支持的是有连接的传输层服务,使用的端口号是 TCP 协议的端口号。客户机是使用网络服务的应用进程,它通过临时端口号向服务器请求服务。服务器是提供网络服务的应用进程,为了使众多的客户机知道服务器的存在,它通过熟知端口号来向客户机提供服务。表 10-1 给出了 TCP 的主要熟知端口号。RFC1700 文档给出了熟知端口号的分配情况。RFC3232 文档是对熟知端口号的增加与补充。

表 10-1　TCP 的主要熟知端口号

端口号	服务进程	说　　明
20	FTP	文件传输协议(数据连接)
21	FTP	文件传输协议(控制连接)
23	Telnet	虚拟终端网络
25	SMTP	简单邮件传输协议
53	DNS	域名系统
80	HTTP	超文本传输协议
110	POP3	邮局协议第 3 版
443	HTTPS	安全超文本传输协议

10.3 例题分析

10.3.1 设计要求

编写程序来扫描服务器已开启的TCP端口,并将获得的相应端口号显示出来。在本练习中为了简便起见,只扫描从0～127范围内的端口。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为ScanPort.exe,则程序的命令行格式为:

```
ScanPort server_addr
```

其中,server_addr为服务器的IP地址。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
已开启的TCP端口: xx xx xx
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

10.3.2 关键问题

1. 创建套接字

在进行网络环境下的Socket编程时,首先需要进行的是为通信创建一个套接字,这时涉及的两个重要函数是:WSAStartup()与socket()。其中,WSAStartup()函数根据请求的Socket版本搜索相应的Socket库,并将找到的Socket库绑定到应用程序,以后就可以异步方式调用其他Socket函数;socket()函数用来创建一个能进行网络通信的套接字。需要注意的是,当应用程序完成本次网络通信后,需要调用closesocket()函数关闭自己创建的套接字,调用WSACleanup()函数解除Socket库绑定并释放占用的资源。

下面给出创建套接字的伪代码:

```
//套接字异步启动
if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)
//创建流式Socket
SOCKET sock=socket(AF_INET,SOCK_STREAM,0);
//填充Socket地址
sockaddr_in serveraddr;
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=端口号;
serveraddr.sin_addr.S_un.S_addr=IP地址;
```

2. 端口扫描

常用的 TCP 端口扫描技术主要包括三种：Connect 扫描、SYN 扫描与 FIN 扫描。其中，Connect 扫描是利用套接字的 connect()函数进行扫描，扫描每个端口都需完成建立完整的 TCP 连接的三次握手过程，这种方式又称为全连接扫描。图 10-5 给出了 Connect 扫描的工作原理。SYN 扫描是利用包含 SYN 标志的 TCP 包进行扫描，扫描每个端口仅需完成建立 TCP 连接的第一次握手，若服务器没开启端口则会返回 RST 包关闭连接，这种方式又称为半连接扫描。FIN 扫描是利用包含 FIN 标志的 TCP 包进行扫描，若服务器开启端口则会丢弃该 TCP 包，若服务器没开启端口则返回 RST 包，这种方式不需要建立 TCP 连接。

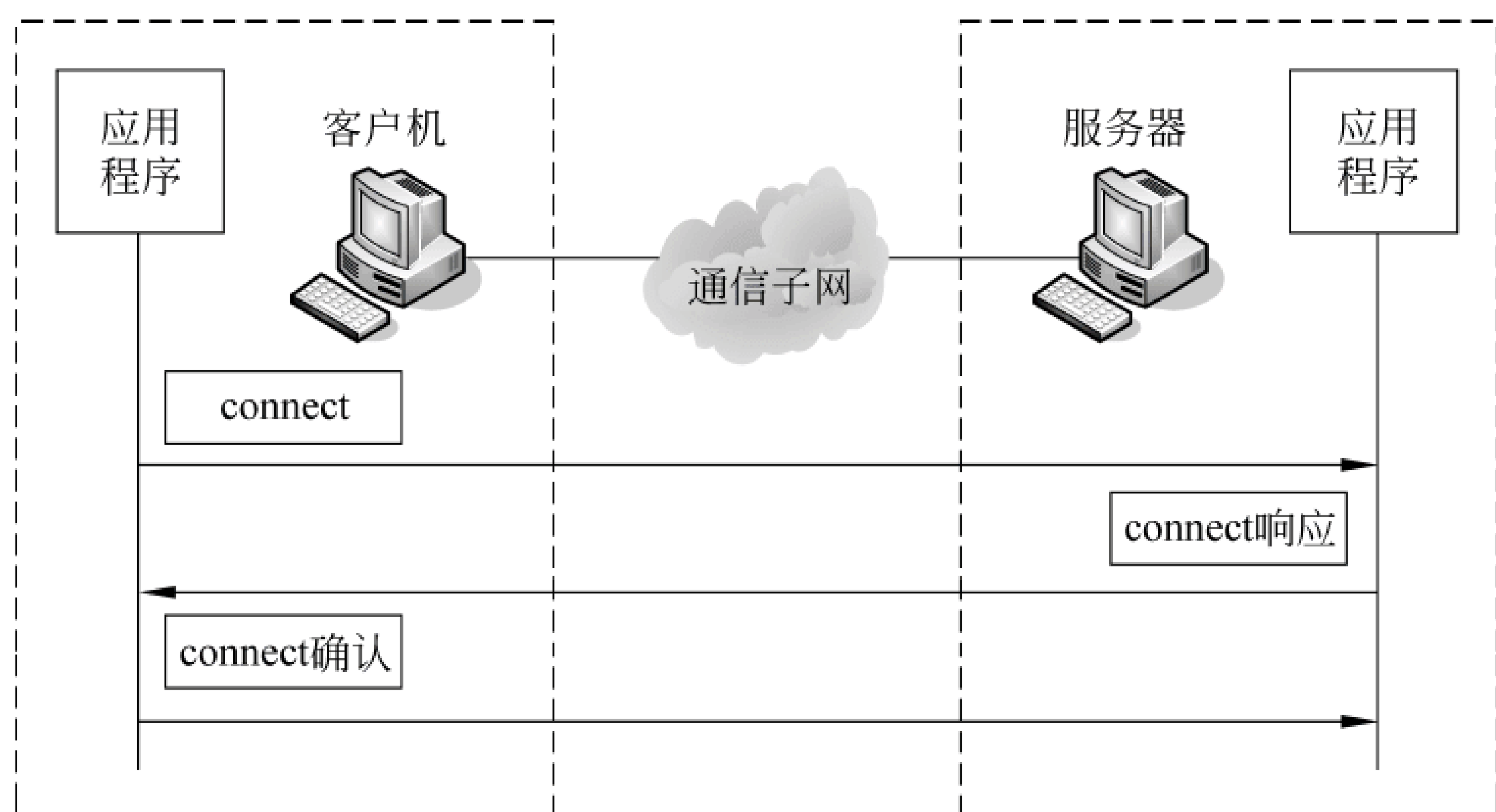


图 10-5　Connect 扫描的工作原理

本次练习采用的是 Connect 扫描方式，利用 connect()函数与某个端口建立连接。如果该端口开启并处于侦听状态，则这次 connect 连接就会成功建立；否则，该端口未开启而不能建立连接。Connect 扫描方式的优点是正常建立 TCP 连接，在编程上可调用 connect()函数来轻松完成。如果在一个进程中依次扫描每个端口，每次调用 connect()函数建立连接造成扫描速度较慢，可以采用多个线程并发执行提高扫描速度。与 Connect 扫描相比，SYN 扫描与 FIN 扫描的执行速度较快，但是在编程实现上更复杂并存在不确定性。

下面给出 Connect 扫描的伪代码：

```
//设置超时时间
struct timeval timeout;
timeout.tv_sec=100/1000;
timeout.tv_usec=0;
//与端口建立连接
connect(sock,&serveraddr,sizeof(serveraddr));
//判断连接是否超时
if(select(0,NULL,&write,NULL,&timeout)>0)
    …
```

3. 程序流程图

图 10-6 给出了主程序流程图。要求输入的命令行参数必须正确，除了程序本身的名称

以外,还需要一个输入文件名作为参数。如果命令行参数的个数不是一个,或者输入文件无法正确打开,则程序在输出错误信息后退出。在主程序的流程中,需要判断是否扫描端口,以及端口是否开启。

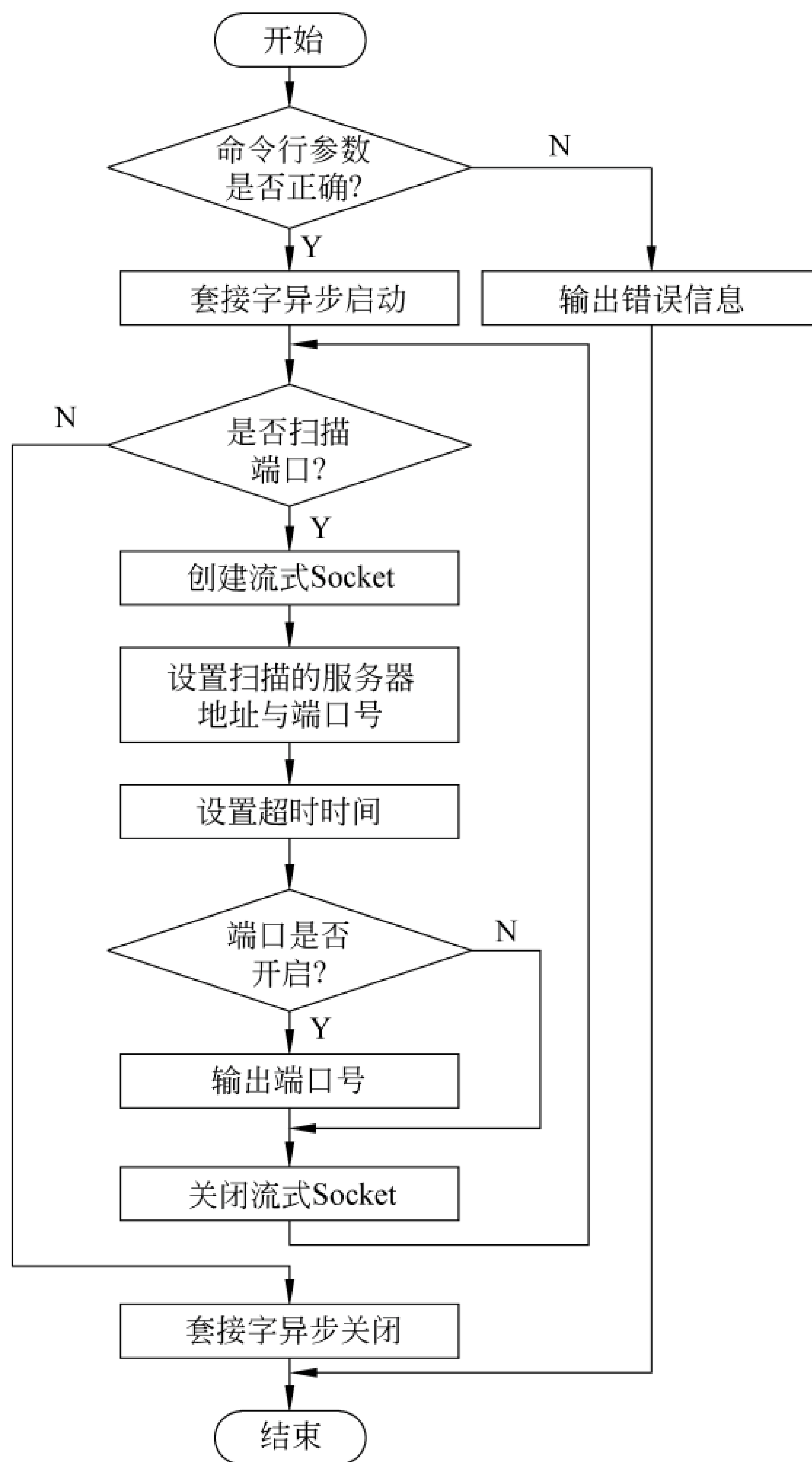


图 10-6　主程序流程图

10.3.3　程序源代码

下面给出端口扫描程序的源代码:

```
//ScanPort.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "winsock.h"
#include "iostream"
using namespace std;
```

```
#pragma comment(lib,"ws2_32")                          //加载 ws2_32.lib

void main(int argc,char* argv[])
{
    if(argc!=2)                                        //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行：ScanPort server_addr"<<endl;
        return;
    }

    WSADATA WSAData;
    if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)          //套接字异步启动
    {
        cout<<endl<<"WSAStartup 初始化失败"<<endl;
        return;
    }
    cout<<endl<<"已开启的 TCP 端口:";

    for(int i=0;i<32;i++)
    {
        SOCKET sock=socket(AF_INET,SOCK_STREAM,0);     //创建原始 Socket
        if(sock==INVALID_SOCKET)
        {
            cout<<endl<<"创建 Socket 失败!"<<endl;
            return;
        }
        else
        {
            sockaddr_in serveraddr;                    //填充 Socket 地址
            serveraddr.sin_family=AF_INET;
            serveraddr.sin_port=htons((unsigned short)i);
            serveraddr.sin_addr.S_un.S_addr=inet_addr(argv[1]);

            int nConnect=connect(sock,(sockaddr*)&serveraddr,sizeof(serveraddr));
            if(nConnect==SOCKET_ERROR)                 //与端口建立连接
                continue;

            struct fd_set write;                       //写 Socket 集合
            FD_ZERO(&write);
            FD_SET(sock,&write);

            struct timeval timeout;                    //设置超时时间
            timeout.tv_sec=100/1000;
            timeout.tv_usec=0;
```

```
            if(select(0,NULL,&write,NULL,&timeout)>0)
            cout<<i<<"  ";                    //判断端口是否打开

            closesocket(sock);                //关闭原始 Socket
        }
    }

    WSACleanup();                             //套接字异步关闭
    cout<<endl<<"TCP端口扫描完成"<<endl;
}
```

图10-7给出了端口扫描程序的执行过程。程序命令行输入为ScanPort 10.134.37.128。端口扫描程序向服务器的0～127端口依次发送连接请求,并将在超时时间内返回响应的端口号显示在控制台上。

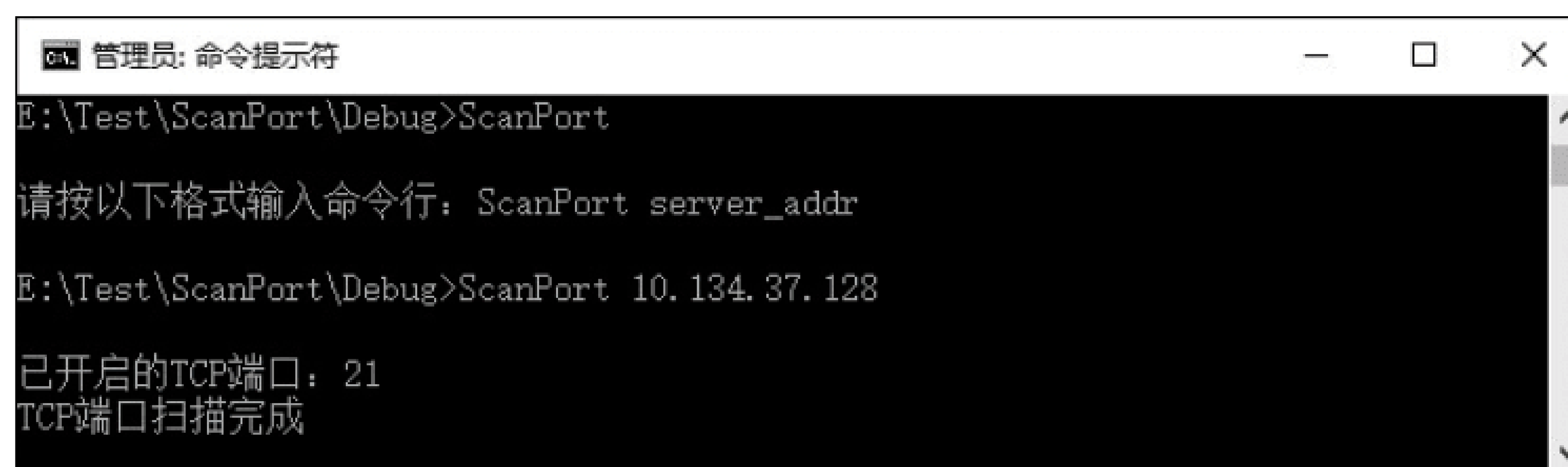


图10-7　端口扫描程序的执行过程

10.4　练　习　题

编写程序来扫描服务器已开启的TCP端口,并将获得的相应端口号显示出来。在本练习中需要扫描从0～1023范围的端口,并采用多线程技术来提高端口扫描速度。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为ScanPort.exe,则程序的命令行格式为:

```
ScanPort server_addr
```

其中,server_addr为服务器的IP地址。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
已开启的TCP端口:xx xx xx
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第11章 TCP数据包的封装与发送

11.1 设计目的

TCP协议是TCP/IP协议族的核心协议之一。熟悉TCP包结构对于理解网络层次结构,以及TCP协议与IP协议的关系有着重要的意义。本章练习的目的是,根据TCP协议的基本原理,通过封装与发送一个标准的TCP数据包,了解TCP包结构中各字段的含义与用途,从而深入理解传输层与下面各层的关系。

11.2 相关知识

本章涉及的相关知识包括TCP协议的概念与TCP数据包结构。

11.2.1 TCP协议的基本概念

传输层的主要作用是实现网络中主机之间的分布式进程通信。传输层协议可以分为两种类型:TCP协议与UDP协议。其中,TCP协议是一种可靠的、面向连接的传输层协议,它允许将源主机的数据无差错地传输到目的主机。UDP协议是一种不可靠的、无连接的传输层协议。TCP协议与UDP协议针对的是不同类型的网络应用。TCP协议的功能是建立在IP协议的基础上,它允许在两个应用进程之间建立一条连接,通过该连接可以实现顺序的、无差错的、不重复的数据流传输。

根据OSI参考模型的定义,传输层使用下面的网络层提供的服务,并且要向上面的应用层提供服务。由于TCP协议所处的层次高于IP协议,因此TCP数据包需要封装在IP分组中传输。图11-1给出了TCP数据包的封装过程。当某种应用进程是基于TCP协议时,首先将应用层数据作为TCP数据与TCP头部封装成TCP数据包,然后将TCP包作为IP数据部分与IP头部封装成IP分组。在将TCP数据包封装成IP分组时,IP头部的协议字段表示上层协议类型,用于表示TCP协议的字段值为6。

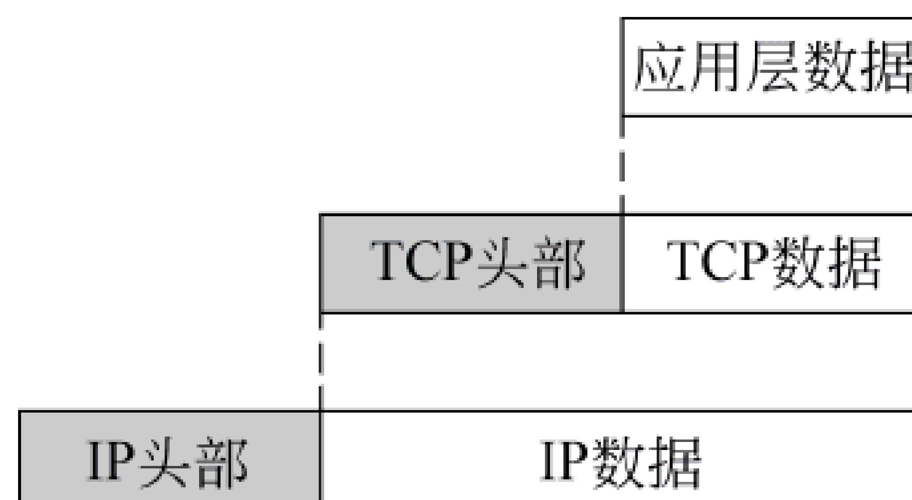


图11-1　TCP数据包的封装过程

在传输层的协议结构中,UDP协议是一种能够满足最低传输要求的协议,而TCP协议

是一种功能完善的传输层协议。面向连接对提高数据传输的可靠性是很重要的。应用程序在使用TCP协议来传输数据之前,必须在通信双方的进程之间建立一个TCP连接。每个TCP连接都要使用通信双方的端口号来标识,并且为双方的一次进程通信提供服务。TCP协议允许通信双方的应用程序在任何时候传输数据,因此通信双方都设置有相应的发送与接收缓冲区,它们分别用于缓存应用程序需发送或接收的数据流。

当客户机与服务器之间的TCP连接建立后,通信双方就可以通过这个连接进行全双工的字节流传输。为了保证TCP协议能够正常、有效地运行,TCP软件设置了一个保持计时器(keep timer),用于防止TCP连接长期处于空闲状态。TCP协议使用以字节为单位的滑动窗口(sliding window)机制,用于控制字节流的发送、接收、确认与重传过程。发送方为发送缓冲区设置一个发送窗口,只要这个窗口值不为0就可以发送字节流。接收方为接收缓冲区设置一个接收窗口,窗口值等于接收缓冲区可继续接收的字节流数。

11.2.2 TCP数据包的结构

设计TCP协议的主要原则是功能全面,数据传输的可靠性有保证。RFC793是最早出现的TCP协议文档,它描述了TCP协议的主要内容。后来,出现了几十种对TCP协议进行扩展与调整的RFC文档。例如,RFC2415是对滑动窗口与确认策略的补充;RFC2581是对拥塞控制机制的补充;RFC2988是对重传计时器的补充。这些RFC文档共同构成了TCP协议功能。图11-2给出了TCP数据包的结构。TCP数据包分为两个部分:TCP头部与TCP数据。TCP头部的长度为20~60B。

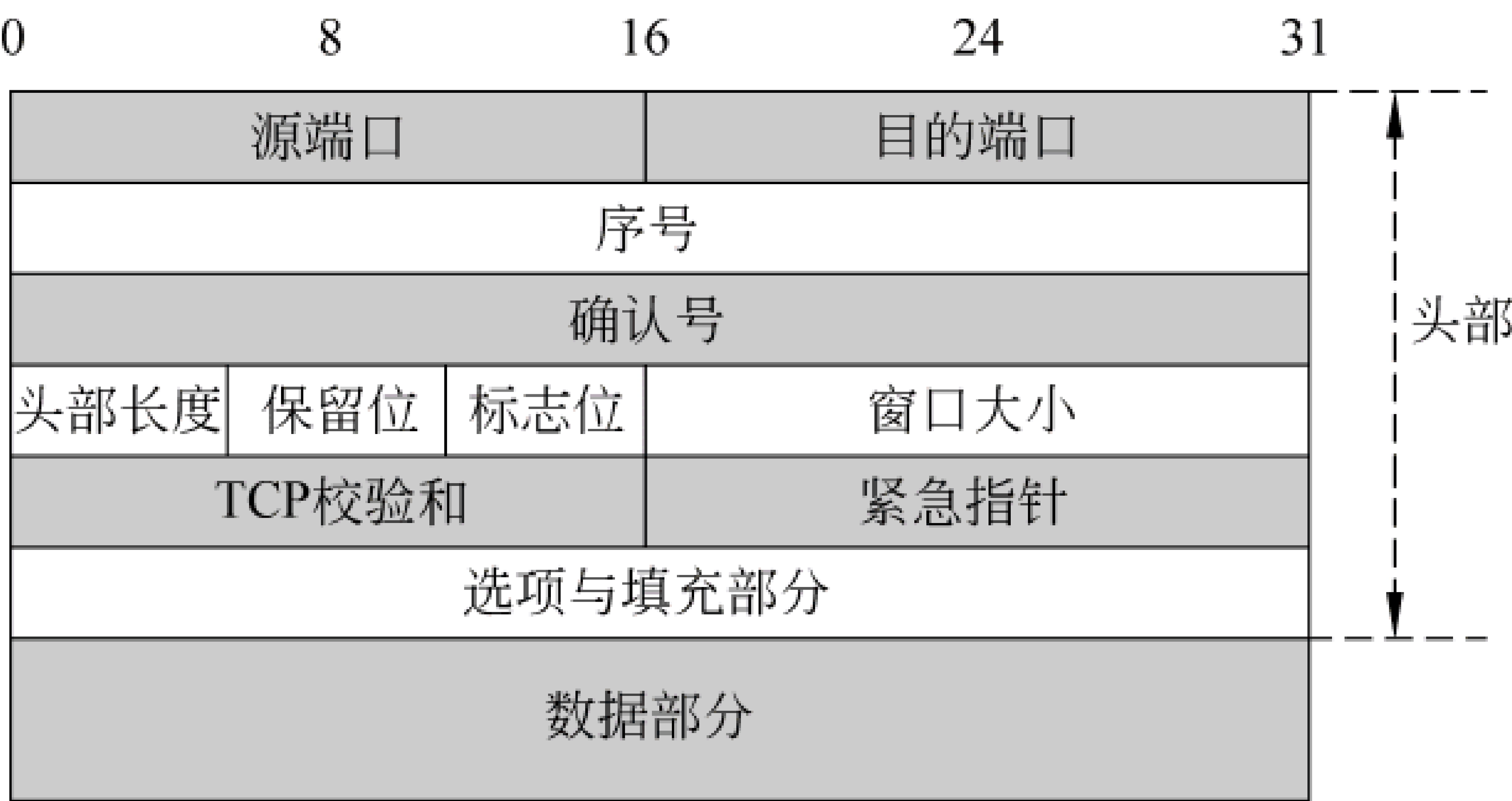


图11-2　TCP数据包的结构

TCP数据包头部由以下字段组成。

1. 端口号

端口号(port number)字段包括两个部分:源端口号(source port number)与目的端口号(destination port number)。这两个字段的长度均为16位。源端口号表示发送方的应用程序使用的TCP端口号,目的端口号表示接收方的应用程序使用的TCP端口号。无论对发送方还是接收方来说,服务器使用的TCP端口是预先规定的熟知端口号,客户机使用的TCP端口是临时申请的临时端口号。

2. 序号

序号(sequence)字段的长度为32位,表示TCP包的第一字节的序号。由于TCP协议是面向数据流的传输层协议,传输的TCP包可被视为一个连续的字节流,因此TCP协议需

要为数据流中的每个字节编号。例如，某个 TCP 包的序号值为 201，携带的数据长度为 100B，则它的第一字节的序号为 201，最后字节的序号为 300。由于通信双方各自随机生成一个初始序号，因此一个 TCP 连接的通信双方的序号不同。

3. 确认号

确认号(acknowledge)字段的长度为 32 位，表示接收方已正确接收序号为 N 的字节，要求发送方接下来发送序号为 $N+1$ 开始的字节流。例如，如果接收方接收到一个序号为 201、长度为 100B 的 TCP 包，则它向发送方发送一个确认号为 301 的 TCP 包。TCP 协议采用的是典型的捎带确认方法。

4. 头部长度

头部长度(header length)字段的长度为 4 位，表示 TCP 数据包的头部长度。TCP 头部中除了选项与填充字段之外，其他字段的长度都是固定的。TCP 头部的基本长度为 20B，如果加上最长 40B 的选项字段，则 TCP 头部的最大长度为 60B。

5. 保留位

保留位(reserved)字段的长度为 6 位，用于保留供以后使用。目前，该字段在使用时所有位设置为 0。

6. 标志位

标志位(flags)字段的长度为 6 位，用于设置 6 种不同的标志位，可以同时设置其中的一位或多位。表 11-1 给出了 6 个标志位的主要用途。发送方将 URG 位设置为 1，表示该数据的优先级较高，需要插入 TCP 数据流的最前面。URG 位需要与紧急指针字段一起使用。SYN 位在连接建立时用于同步序号。例如，连接建立请求的 SYN＝1、ACK＝0，连接建立响应的 SYN＝1、ACK＝1。在连接建立后，所有 TCP 数据包的 ACK 位用于捎带确认。

表 11-1　6 个标志位的主要用途

标　志　位	用　　途
紧急(URG)	数据的优先级高
确认(ACK)	数据的确认号有效
推送(PSH)	数据希望尽快获得响应
复位(RST)	在出现问题时，强制释放连接
同步(SYN)	在连接建立时，同步序号
终止(FIN)	在正常情况下，释放连接

7. 窗口大小

窗口大小(window size)字段的长度为 16 位，表示以字节(B)为单位的窗口大小。窗口大小的最大值为 $2^{16}-1$，即 65 535B。由于接收方的接收缓冲区是受限制的，因此用窗口字段表示接收方还有多大的接收容量。发送方会根据接收方通知的窗口值，以便调整自己的发送窗口值的大小。窗口大小字段值是动态变化的。

8. TCP 校验和

TCP 校验和(check sum)字段的长度为 16 位，用于检查 TCP 包在传输中是否出错，其计算方法与 IP 头部校验和的计算方法相同。TCP 校验和字段的校验范围是：伪头部、TCP 头部与 TCP 数据。伪头部(pseudo header)的长度为 12B，它本身并不是 TCP 包的真正头部，只是在计算校验和时临时与 TCP 包相连。图 11-3 给出了 TCP 伪头部的结构。伪头部

内容主要来自IP分组头部的一部分,它包括源IP地址(16位)、目的IP地址(16位)、协议(8位)与UDP长度(16位)字段。



图11-3 TCP伪头部的结构

9. 紧急指针

紧急指针(urgent point)字段的长度为16B,表示TCP包中有紧急数据需要发送。当URG位的值为1时,紧急指针字段才能生效。在优先处理紧急数据之后,TCP协议软件才能够回复正常操作。

10. 选项

选项(options)字段的长度范围是0～40B。选项字段包括两类:单字节选项与多字节选项。单字节选项有两项:选项结束与无操作。多字节选项有三项:最大报文段长度、窗口扩大因子与时间戳。在使用选项字段时,如果TCP头部长度不是32位的整数倍,这时就需要通过填充位(0)来凑齐。

11.3 例题分析

11.3.1 设计要求

根据协议规定的TCP数据包的标准格式,编写程序构造TCP包结构与填写各个字段,并将封装后的TCP包内容写入输出文件。在本练习中为了简便起见,数据字段通过为字符串赋值来获得,但是需要计算相应的头部校验和。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为TcpEncap.exe,则程序的命令行格式为:

```
TcpEncap output_file
```

其中,output_file为输出文件的名称。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
IP头部字段
总长度:xx
IP校验和:xx
源 IP地址:xx.xx.xx.xx
目的 IP地址:xx.xx.xx.xx
TCP头部与数据字段
TCP长度:xx
源端口:xx
目的端口:xx
```

```
TCP 校验和:xx
数据字段:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

11.3.2 关键问题

1. 定义 TCP 头部与伪头部的数据结构

TCP 头部与 TCP 数据需要作为 IP 数据,与 IP 头部封装成 IP 数据包后才能发送。在对 TCP 头部字段进行填充之前,需要构造 TCP 头部的数据结构,该数据结构与图 11-2 的 TCP 包头部结构一致;然后,需要构造伪头部的数据结构,伪头部只是临时用来为 TCP 包计算校验和,并不需要作为 TCP 包的一部分来发送。另外,还需要构造 IP 头部的数据结构,这部分已经在第 7 章中介绍过。

下面给出构造 TCP 头部与伪头部的伪代码:

```
//定义 TCP 头部结构
typedef struct TCP_HEAD
{
    unsigned short SourcePort;
    unsigned short DestinPort;
    unsigned int Sequence;
    unsigned int Acknowledge;
    union
    {
        unsigned short HeadLen;
        unsigned short Reserved;
        unsigned short Flags;
    };
    unsigned short WindowsLen;
    unsigned short TcpChecksum;
    unsigned short UrgePoint;
}tcp_head;
//定义 TCP 伪头部结构
typedef struct PSD_HEAD
{
    unsigned int SourceAddr;
    unsigned int DestinAddr;
    unsigned char Reserved;
    unsigned char Protocol;
    unsigned short TcpLen;
}psd_head;
```

2. 填充数据包与计算校验和

在填充TCP数据包的过程中,需要分别填充IP头部、TCP头部与TCP数据。为了填充TCP头部的校验和字段,还需要填充TCP头部附加的伪头部。IP头部与TCP头部的校验和需要分别计算。首先,IP头部的校验和字段赋初值0,调用校验和计算函数checksum()对IP头部进行校验;然后,TCP头部的校验和字段赋初值0,调用校验和计算函数checksum()对TCP头部与伪头部进行校验。

下面给出填充TCP头部与伪头部的伪代码:

```
//初始化相关对象
psd_head psd={0};
tcp_head tcp={0};
unsigned short check[65535];
const char tcp_data[]={"This is a test of tcp packet encapsule!"};
//填充 TCP 伪头部字段
psd.SourceAddr=ip.SourceAddr;
psd.DestinAddr=ip.DestinAddr;
psd.Reserved=0;
psd.Protocol=ip.Protocol;
psd.TcpLen=sizeof(tcp_head)+sizeof(tcp_data);
//填充 TCP 头部字段
tcp.SourcePort=1000;
tcp.DestinPort=1000;
tcp.Sequence=0;
tcp.Acknowledge=0;
tcp.HeadLen=(sizeof(tcp_head)/sizeof(unsigned int)<<4|0);
tcp.WindowsLen=10000;
tcp.TcpChecksum=0;
tcp.UrgePoint=0;
//计算 TCP 包(包括伪头部)的校验和
memset(check,0,65535);
memcpy(check,&psd,sizeof(psd_head));
memcpy(check+sizeof(psd_head),&tcp,sizeof(tcp_head));
memcpy(check+sizeof(psd_head)+sizeof(tcp_head),tcp_data,sizeof(tcp_data));
tcp.TcpChecksum=checksum(check,sizeof(psd_head)+sizeof(tcp_head)+sizeof(tcp_data));
```

3. 程序流程图

图11-4给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要有一个输出文件名。如果命令行参数的个数不是一个,则程序在输出错误信息后退出。

11.3.3 程序源代码

下面给出TCP包封装程序的源代码:

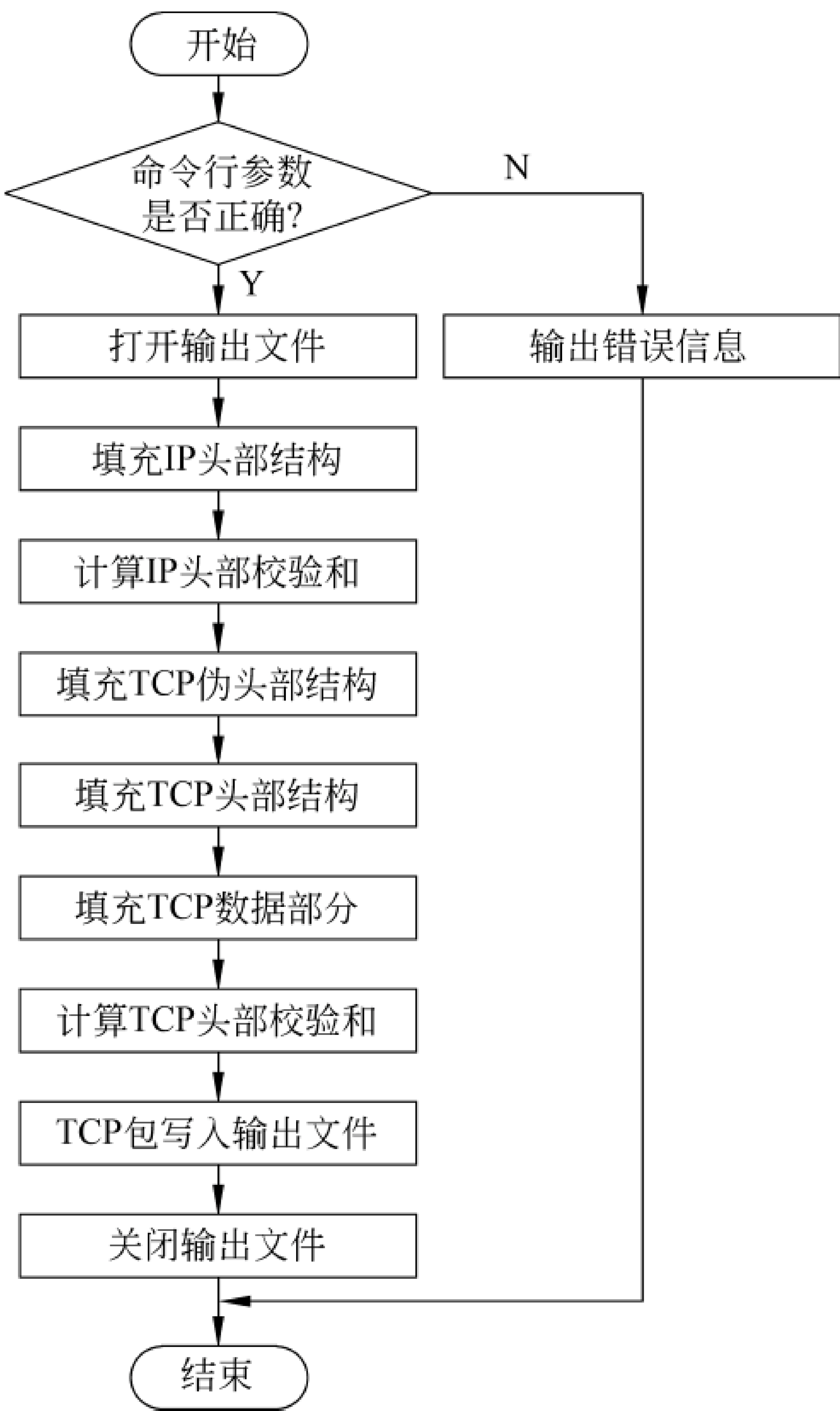


图 11-4　主程序流程图

```
//TcpEncap.cpp：定义控制台应用程序的入口点

#include "stdafx.h"
#include "winsock2.h"
#include "fstream"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib

typedef struct IP_HEAD                           //定义 IP 头部结构
{
    union
    {
        unsigned char Version;                   //版本（字节前 4 位）
        unsigned char HeadLen;                   //头部长度（字节后 4 位）
    };
    unsigned char ServiceType;                   //服务类型
    unsigned short TotalLen;                     //总长度
```

```
    unsigned short Identifier;                          //标识符
    union
    {
        unsigned short Flags;                           //标志位(字前3位)
    unsigned short FragOffset;                          //片偏移(字后13位)
    };
    unsigned char TimeToLive;                           //生存周期
    unsigned char Protocol;                             //协议
    unsigned short HeadChecksum;                        //头部校验和
    unsigned int SourceAddr;                            //源 IP 地址
    unsigned int DestinAddr;                            //目的 IP 地址
}ip_head;

typedef struct PSD_HEAD                                 //定义 TCP 伪头部结构
{
    unsigned int SourceAddr;                            //源 IP 地址
    unsigned int DestinAddr;                            //目的 IP 地址
    unsigned char Reserved;                             //保留位
    unsigned char Protocol;                             //协议
    unsigned short TcpLen;                              //TCP 长度
}psd_head;

typedef struct TCP_HEAD                                 //定义 TCP 头部结构
{
    unsigned short SourcePort;                          //源端口
    unsigned short DestinPort;                          //目的端口
    unsigned int Sequence;                              //序列号
    unsigned int Acknowledge;                           //确认号
    union
    {
        unsigned short HeadLen;                         //头部长度(字前4位)
        unsigned short Reserved;                        //保留位(字中6位)
        unsigned short Flags;                           //标志位(字后6位)
    };
    unsigned short WindowsLen;                          //窗口大小
    unsigned short TcpChecksum;                         //TCP 校验和
    unsigned short UrgePoint;                           //紧急指针
}tcp_head;

unsigned short check[65535];                            //设置校验缓冲区
const char tcp_data[]={"This is a test of tcp packet encapsule!"};

unsigned short checksum(unsigned short * buffer,int size)
{                                                       //校验和计算函数
```

```
    unsigned long cksum=0;
    while(size>1)
    {
        cksum+=*buffer++;
        size-=sizeof(unsigned short);
    }
    if(size)
        cksum+=*(unsigned char*)buffer;
    cksum=(cksum>>16)+(cksum&0xffff);
    cksum+=(cksum>>16);
    return(unsigned short)(~cksum);
}

void main(int argc,char* argv[])
{
    if(argc!=2)                                    //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行：TcpEncap output_file"<<endl;
        return;
    }

    fstream outfile;                               //创建输出文件流
    outfile.open(argv[1],ios::in|ios::out);        //打开输出文件

    //填充 IP 包头部各字段
    ip_head ip={0};
    ip.Version=(0x04<<4|sizeof(ip_head)/sizeof(unsigned int));
    ip.ServiceType=0;
    ip.TotalLen=sizeof(ip_head)+sizeof(tcp_head)+sizeof(tcp_data);
    ip.Identifier=0;
    ip.TimeToLive=128;
    ip.Protocol=IPPROTO_TCP;
    ip.HeadChecksum=0;
    ip.SourceAddr=inet_addr("192.168.1.15");
    ip.DestinAddr=inet_addr("192.168.1.22");

    //计算 IP 包头部各字段的校验和
    memset(check,0,65535);
    memcpy(check,&ip,sizeof(ip_head));
    ip.HeadChecksum=checksum(check,sizeof(ip_head));

    //显示 IP 头部的部分字段值
    cout<<endl<<"IP 头部字段"<<endl;
    cout<<"总长度:"<<ip.TotalLen<<endl;
    cout<<"IP 校验和:"<<ip.HeadChecksum<<endl;
    cout<<"源 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.SourceAddr)<<endl;
```

```
    cout<<"目的 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.DestinAddr)<<endl;

    //填充 TCP 伪头部各字段
    psd_head psd={0};
    psd.SourceAddr=ip.SourceAddr;
    psd.DestinAddr=ip.DestinAddr;
    psd.Reserved=0;
    psd.Protocol=ip.Protocol;
    psd.TcpLen=sizeof(tcp_head)+sizeof(tcp_data);

    //填充 TCP 头部各字段
    tcp_head tcp={0};
    tcp.SourcePort=1000;
    tcp.DestinPort=1000;
    tcp.Sequence=0;
    tcp.Acknowledge=0;
    tcp.HeadLen=(sizeof(tcp_head)/sizeof(unsigned int)<<4|0);
    tcp.WindowsLen=htons((unsigned short)10000);
    tcp.TcpChecksum=0;
    tcp.UrgePoint=0;

    //计算 TCP 包(包括伪头部与数据)的校验和
    memset(check,0,65535);
    memcpy(check,&psd,sizeof(psd_head));
    memcpy(check+sizeof(psd_head),&tcp,sizeof(tcp_head));
    memcpy(check+sizeof(psd_head)+sizeof(tcp_head),tcp_data,sizeof(tcp_data));
    tcp.TcpChecksum=checksum(check,sizeof(psd_head)+sizeof(tcp_head)+
    sizeof(tcp_data));

    //显示 TCP 头部的部分字段与数据部分
    cout<<endl<<"TCP 头部与数据字段"<<endl;
    cout<<"TCP 长度:"<<psd.TcpLen<<endl;
    cout<<"源端口:"<<tcp.SourcePort<<endl;
    cout<<"目的端口:"<<tcp.DestinPort<<endl;
    cout<<"TCP 校验和:"<<tcp.TcpChecksum<<endl;
    cout<<"数据字段:"<<tcp_data<<endl;

    //依次写入 IP 头部、伪头部、TCP 头部与数据
    outfile.write((char*)&ip,sizeof(ip_head));
    outfile.write((char*)&tcp,sizeof(tcp_head));
    outfile.write(tcp_data,sizeof(tcp_data));

    cout<<endl<<"TCP 包封装完成"<<endl;
    outfile.close();                              //关闭输出文件
}
```

图 11-5 给出了 TCP 数据包的封装过程。程序命令行输入为 TcpEncap output。程序构造 IP 头部、TCP 头部与伪头部的数据结构，然后依次填充 IP 头部、TCP 头部的字段与数据部分，最后将填充好的 TCP 数据包内容写入 output。

```
命令提示符
E:\Test\TcpEncap\Debug>TcpEncap

请按以下格式输入命令行：TcpEncap output_file

E:\Test\TcpEncap\Debug>TcpEncap output

IP头部字段
总长度：80
IP校验和：33383
源IP地址：192.168.1.15
目的IP地址：192.168.1.22

TCP头部与数据字段
TCP长度：60
源端口：1000
目的端口：1000
TCP校验和：1448
数据字段：This is a test of tcp packet encapsule!

TCP包封装完成
```

图 11-5　TCP 数据包的封装过程

11.4　练　习　题

根据协议规定的 TCP 数据包的标准格式，编写程序构造 TCP 包结构与填写各个字段，并将封装后的 TCP 包发送到目的节点。在本练习中为了简便起见，数据字段通过为字符串赋值来获得，但是需要计算相应的头部校验和。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 TcpSend.exe，则程序的命令行格式为：

```
TcpSend source_addr dest_addr
```

其中，source_addr 为源主机的 IP 地址，dest_addr 为目的主机的 IP 地址。

(2) 要求将部分字段内容显示在控制台上，具体格式为：

```
源 IP 地址:xx.xx.xx.xx
源端口:xx
目的 IP 地址:xx.xx.xx.xx
目的端口:xx
数据字段:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 12 章 基于 TCP 的客户机/服务器程序

12.1 设计目的

网络服务是以客户机/服务器模式工作的，服务器在某些特定端口上提供网络服务，等待客户机发送服务请求，并且进行响应。TCP 服务是需要建立连接的网络服务类型。本章练习的目的是，通过基于 TCP 的客户机与服务器程序设计，了解 TCP 协议的基本概念与主要功能，掌握这类网络应用的设计思路与编程方法。

12.2 相关知识

本章涉及的相关知识包括 TCP 协议的特点与客户机/服务器编程方法。

12.2.1 TCP 协议的主要特点

TCP 协议是一种面向连接、可靠的传输层协议。从应用层的角度来看，TCP 协议在网络层 IP 协议的基础上，通过在两个应用进程之间预先建立连接，为应用层的程序提供可靠的数据流传输服务。图 12-1 给出了 TCP 协议与其他协议的关系。也就是说，TCP 协议为上面的应用层协议提供数据传输服务。TCP 主要用于对传输可靠性要求高的应用层协议。例如，文件传输协议(FTP)、超文本传输协议(HTTP)、简单邮件传输协议(SMTP)、远程登录(Telnet)等。另外，域名系统(DNS)可依赖于 TCP 或 UDP 协议。

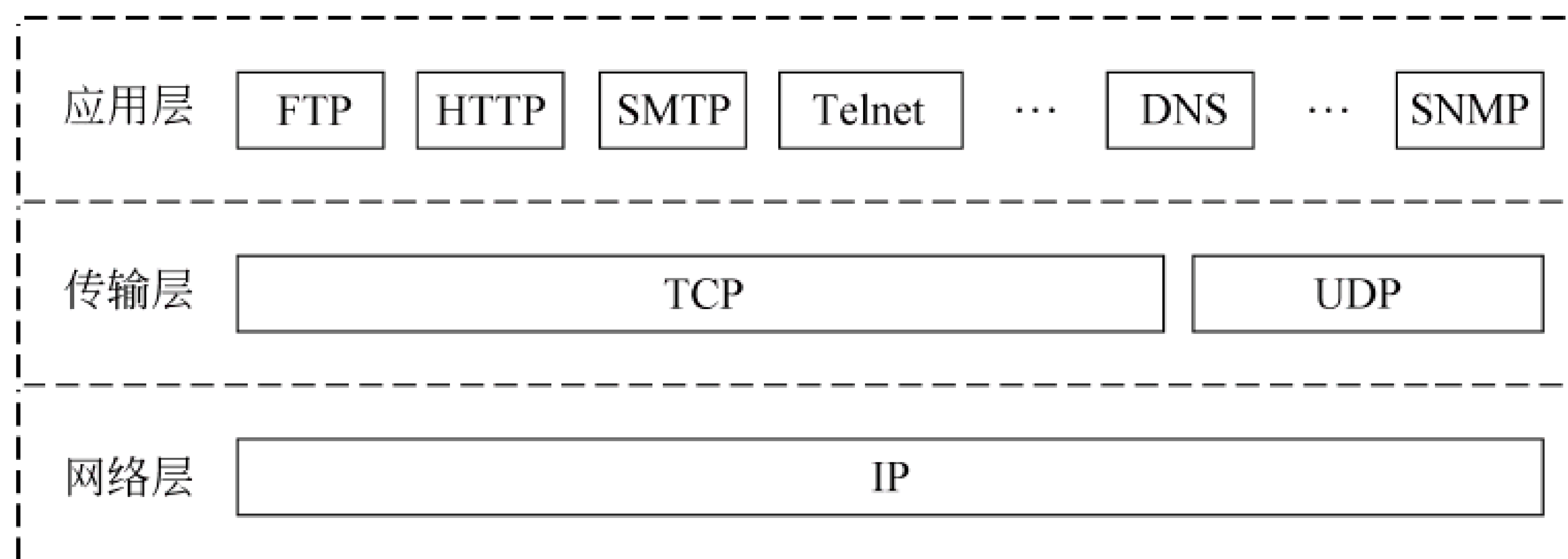


图 12-1　TCP 协议与其他协议的关系

TCP 协议的第一个特征是面向连接。通信双方在传输之前需要建立连接，在传输数据

过程中需要维护连接，在传输结束后需要释放连接。图 12-2 给出了建立 TCP 连接的过程。服务器始终处于侦听状态，检查是否有客户机的连接请求。在建立连接的过程中，客户机与服务器之间经历三次握手：①客户机首先向服务器发送连接请求；②服务器响应连接请求并向客户机返回响应；③客户机再向服务器发送对该响应的确认。建立连接的发起者是客户机。在释放连接的过程中，客户机与服务器之间经历 4 次握手，涉及客户机与服务器两侧的释放请求与响应。释放连接的发起者可以是客户机或服务器。

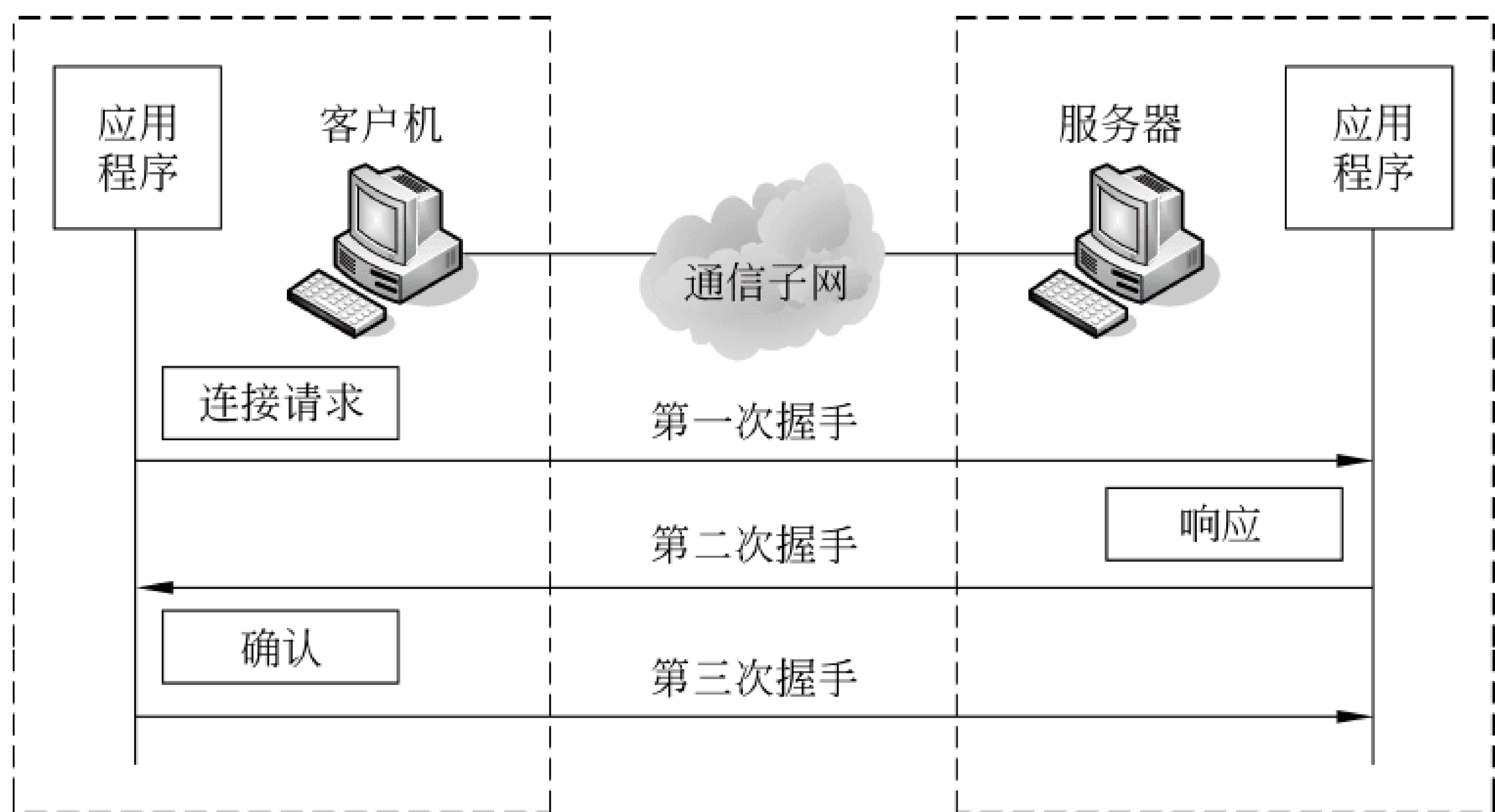


图 12-2　建立 TCP 连接的过程

TCP 协议的第二个特征是数据流传输。流(stream)相当于一个管道，从一端放入的内容可以从另一端原样取出，它描述了一个不出现丢失、重复与乱序的数据传输过程。应用程序与 TCP 协议每次交互的数据长度可能不同，但 TCP 协议是将应用程序提交的数据看成一连串、无结构的字节流。为了能够提供字节流方式的传输，发送方和接收方都需要使用缓存。发送方使用发送缓存来存储应用程序送来的数据。发送方不可能为每个写操作创建一个报文段，而是将几个写操作组合成一个报文段，然后提交给网络层 IP 协议来处理。接收方将接收的数据存储在接收缓存中，应用程序使用读操作从缓存中读出相应的数据。

TCP 协议的第三个特征是可靠的传输服务。TCP 协议通过确认机制来检查数据是否安全到达，关键是对发送和接收的数据进行跟踪、确认与重传。TCP 协议建立在不可靠的网络层 IP 协议之上，当 IP 协议及以下层出现传输错误时，TCP 协议只能不断地进行重传，试图弥补传输中出现的问题。TCP 协议通过校验和计算来检查数据是否正确到达，这个计算过程与 IP 协议的设计思路相同。TCP 协议使用窗口机制来进行流量与拥塞控制。窗口是指通信双方分配的存储接收数据的缓冲区，窗口大小由通信双方在建立连接时协商，但是接收方可以根据需要来动态调整窗口大小。

TCP 协议需要支持同时建立多个连接，这个特点在服务器上表现得更为突出。根据应用程序的需要，TCP 协议支持一个服务器与多个客户机同时建立多个连接，也支持一个客户机与多个服务器同时建立多个连接。TCP 软件将分别管理多个 TCP 连接。在理论上，TCP 协议可以支持同时建立几百甚至上千条这样的连接，但是建立并发连接的数量越多，每条连接所能够共享的资源就会越少。从上述分析可以看出，TCP 协议是一种很复杂的传输层协议，与仅支持简单报文传输的 UDP 协议相比，TCP 可以提供人们所能想到的几乎所有的传输层功能。目前，大多数互联网应用在传输层使用 TCP 协议。

12.2.2 客户机/服务器编程

基于TCP的网络应用采用客户机/服务器模式。客户机是使用网络服务的应用进程,服务器是提供网络服务的应用进程。图12-3给出了基于TCP的客户机/服务器结构。在网络环境中,客户机发送服务请求完全是随机的,并且同时可能有多个客户机发送请求。因此,服务器需要随时在熟知端口侦听请求,并且具备同时处理并发请求的能力,这是服务器与客户机设计中的最大区别。服务器并发处理的解决方案分为两种:并发服务器(concurrent server)与重复服务器(interactive server)。其中,并发服务器使用工作在后台的守护进程(daemon),当有服务请求到达时激活该进程来处理。

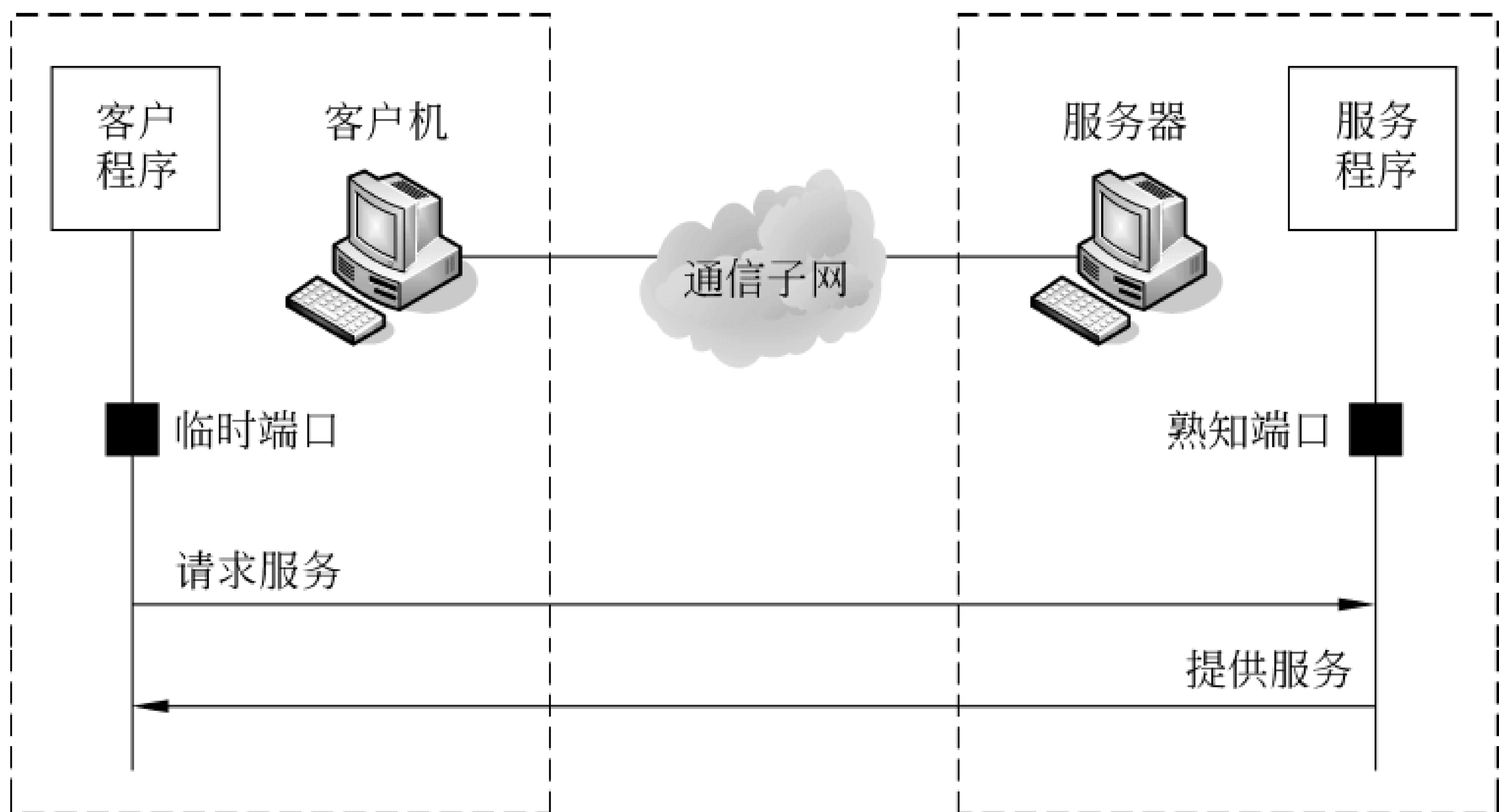


图12-3 基于TCP的客户机/服务器结构

并发服务器本身始终要处于等待并侦听的状态。当服务器接收到客户机发送的服务请求时,它根据该请求的进程号去激活子进程来提供服务,而服务器自身会回到等待状态继续侦听。这里,并发服务器自身被称为主服务器(master),它激活的子进程被称为从服务器(slaver)。主服务器要使用一个全网熟知的进程地址。图12-4给出了并发服务器的工作原理。由于不同从服务器可以并发、独立地处理不同客户机的请求,因此并发服务器更适合于

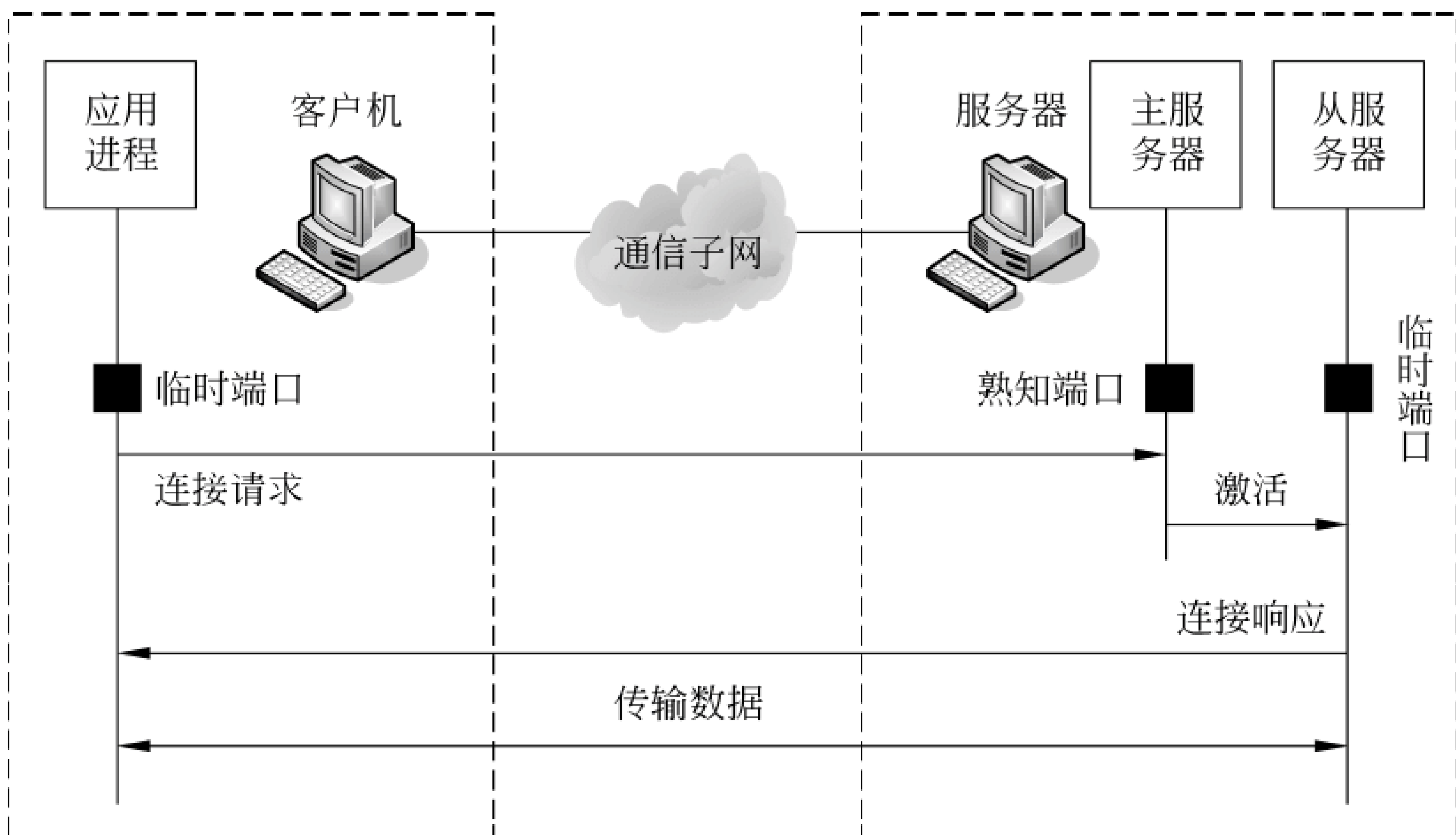


图12-4 并发服务器的工作原理

面向连接的应用类型，也就是基于 TCP 的服务器程序。

无论是客户机还是服务器进程，它们都要通过 IP 地址与端口号来加以标识。在基于 TCP 的网络应用中，使用的端口号是 TCP 协议的端口号。客户机是使用网络服务的应用进程，它通过临时端口号向服务器请求服务。服务器是提供网络服务的应用进程，为了要使众多的客户机知道服务器的存在，它通过熟知端口号来向客户机提供服务。表 12-1 给出了 TCP 的主要熟知端口号。这种熟知端口号(0～1023)是由 IANA 来统一分配的，每个客户机都知道相应服务器的熟知端口号。

表 12-1 TCP 协议的主要熟知端口号

端口号	服务进程	说明
20	FTP	文件传输协议(数据连接)
21	FTP	文件传输协议(控制连接)
23	Telnet	远程登录
25	SMTP	简单邮件传输协议
53	DNS	域名系统
80	HTTP	超文本传输协议
110	POP3	邮局协议第 3 版
443	HTTPS	安全超文本传输协议

12.3 例题分析

12.3.1 设计要求

根据基于 TCP 的客户机/服务器工作模式，编写服务器程序接收客户机的命令，并根据命令向客户机做出响应。客户机向服务器发送 sendfile 命令，服务器向客户机返回 command ok 响应，客户机向服务器发送指定的数据。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 TcpServer.exe，则程序的命令行格式为：

```
TcpServer server_port
```

其中，server_port 为服务器侦听的 TCP 端口。

(2) 要求将服务器的状态显示在控制台上，具体格式为：

```
TCP Server 开始侦听 xx 端口
TCP Server 与 TCP Client 建立连接
TCP Server 接收数据：…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

12.3.2 关键问题

1. 基本编程模式分析

基于TCP的客户机/服务器进程有相对固定的编程模式。如果客户机与服务器进程之间通信,需要依次调用Socket提供的不同函数来实现。但是,服务器编程比客户机编程更复杂。服务器采用重复服务器方式处理多个服务请求。图12-5给出了基于TCP的客户机/服务器编程模式。其中,客户机首先调用socket()函数建立套接字,然后调用connect()函数请求与服务器建立连接,在连接建立后,可以调用send()函数或recv()函数发送或接收数据,最后调用closesocket()函数关闭套接字。

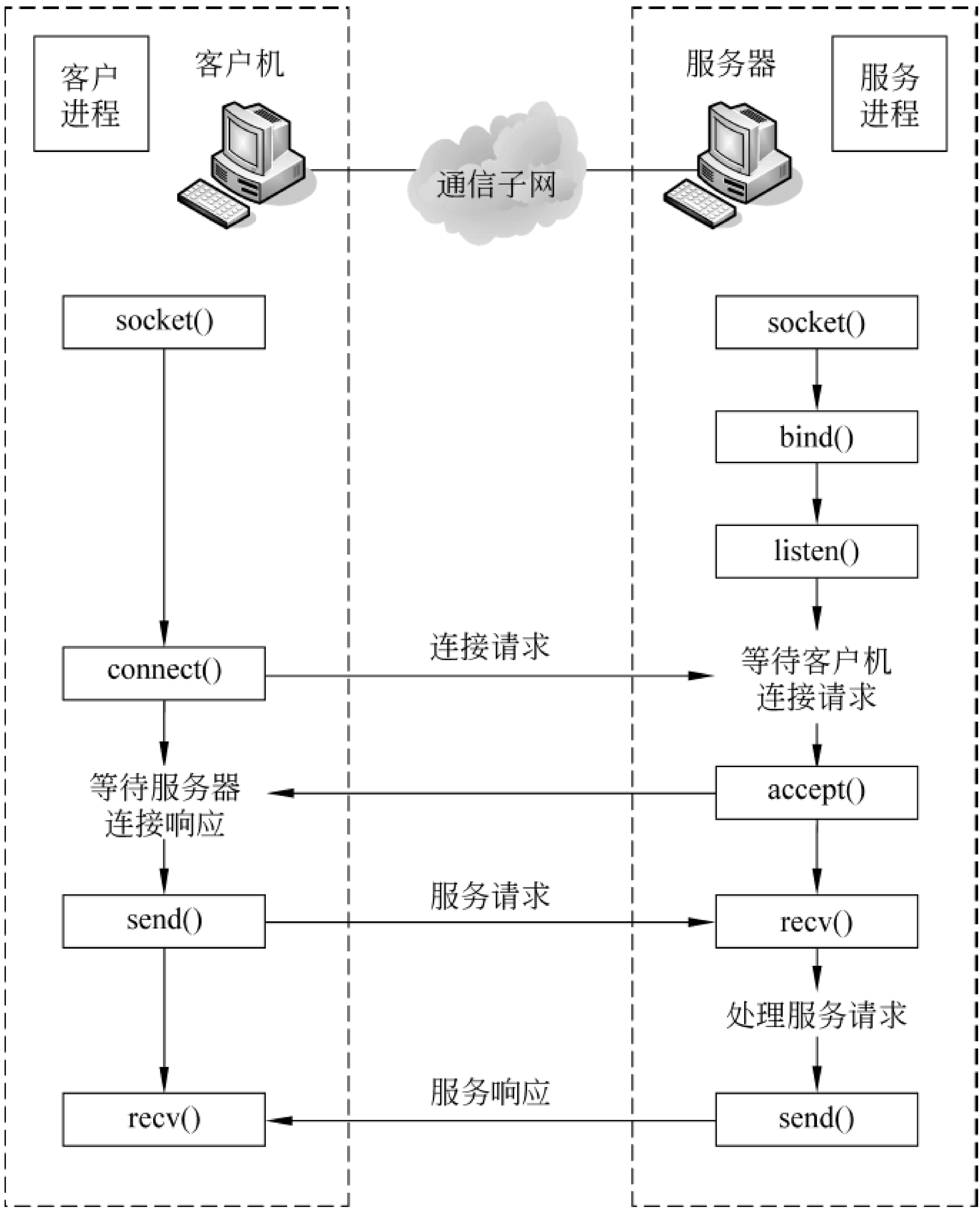


图12-5 TCP客户机/服务器编程模式

服务器首先调用socket()函数建立套接字,然后调用bind()函数将某个端口与套接字绑定,再调用listen()函数在相应端口上侦听连接建立请求。当服务器侦听到有连接建立请求到达时,调用accept()函数创建新临时套接字与客户机建立连接,同时服务器使用原有套接字返回侦听状态。服务器使用新创建子线程与客户机建立连接,这样可以并发处理多个客户机的服务请求。在连接建立后,可以调用send()函数发送数据,或者调用recv()函数接收数据。最后,调用closesocket()函数关闭所有套接字。

2. 创建流式套接字

为了实现基于 TCP 的客户机/服务器进程，首先需要调用 socket()函数创建套接字，其中的 SOCK_STREAM 表示创建流式套接字，IPPROTO_IP 表示采用 IP 协议。接着，调用 bind()函数将某个端口与套接字绑定，调用 listen()函数在相应端口上侦听连接请求。这里，需要使用 INADDR_ANY 来获得本地主机的 IP 地址，然后将 IP 地址与端口号共同填充本地 Socket 结构。

下面给出创建流式套接字的伪代码：

```
//创建流式 Socket
socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
//填充本地 Socket 地址
sockaddr_in serveraddr;
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons((unsigned short)atoi(argv[1]));
serveraddr. sin_addr.S_un.S_addr=htonl(INADDR_ANY);
//将端口与 IP 地址绑定
bind(sock,(sockaddr * )&serveraddr,sizeof(serveraddr));
//在端口上侦听连接
listen(sock,SOMAXCONN);
```

3. 与客户机建立并发连接

由于服务器需要并发处理多个连接请求，因此需要在一个主循环中接收客户机请求，并创建新的服务线程与客户机建立连接。服务器侦听到连接建立请求到达时，调用 accept()函数创建临时套接字与客户机建立连接，同时服务器使用原有套接字继续侦听。在服务器进程执行完毕后，需要调用 closesocket()函数释放所有的套接字。

下面给出与客户机建立并发连接的伪代码：

```
//创建保存线程参数的数据结构
struct ThreadParam
{
    SOCKET sock;
    sockaddr_in addr;
};
//在主循环中接收客户机的连接并创建服务线程
while(true)
{
    //接受客户机的连接请求
    tempsock=accept(sock,(sockaddr * )&tempaddr,&templen);
    //当线程数达到上限，停止接受客户机连接
    if(ThreadCount>=10)
    {
        closesocket(tempsock);
        continue;
```

```
    }
    //设置传递给线程参数
    ThreadParam Param;
    Param.sock=tempsock;
    Param.addr=tempaddr;
    //为每个客户机创建服务线程
    DWORD dwThreadId;
    CreateThread(NULL,0,ServerThread,&Param,0,&dwThreadId);
}
```

4. 在线程中发送与接收数据

当在主循环中接受客户机的连接请求后,需要创建一个服务线程来与客户机建立连接,这时首先要获得由accept()函数返回的临时套接字。当服务线程与客户机建立连接后,服务器可以调用send()函数发送数据,或者调用recv()函数接收数据。当服务器接收到客户机发送的sendfile命令时,需要向客户机返回command ok响应,然后等待接收来自客户机的数据部分。当然,在某个服务线程执行完毕后,需要调用closesocket()函数来释放临时套接字。

下面给出在线程中发送与接收数据的伪代码:

```
DWORD WINAPI ServerThread(LPVOID lpParam)
{
    //从线程参数中获得临时套接字
    SOCKET tempsock=((ThreadParam*)lpParam)->sock;
    sockaddr_in tempaddr=((ThreadParam*)lpParam)->addr;
    //通过端口接收客户机命令
    recv(tempsock,recvbuf,sizeof(recvbuf),0);
    if(strcmp(recvbuf,"sendfile")!=0)
        …
    //通过端口向客户机返回响应
    send(tempsock,"command ok",sizeof("command ok"),0);
    //通过端口接收客户机的数据
    recv(tempsock,recvbuf,sizeof(recvbuf),0);
    //关闭临时套接字
    closesocket(tempsock);
}
```

5. 程序流程图

图12-6给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要提供一个服务器端口号。如果命令行参数的个数不是一个,则程序在输出错误信息后退出。在主程序的流程中,需要判断是否有线程在执行,以及线程数是否达到上限。图12-7给出了子线程流程图。在子线程的流程中,需要判断命令格式是否正确以及数据接收是否结束。

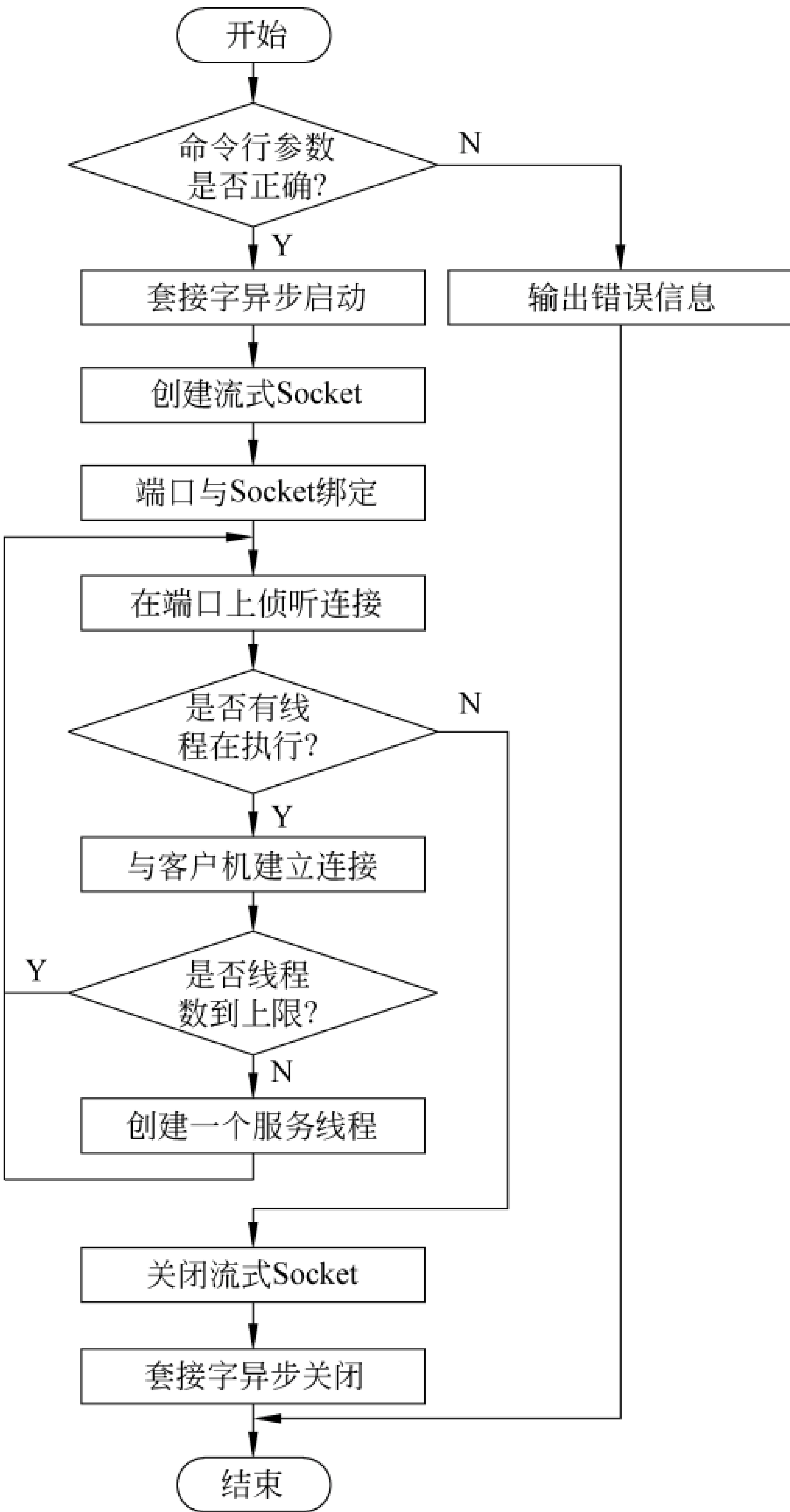


图 12-6　主程序流程图

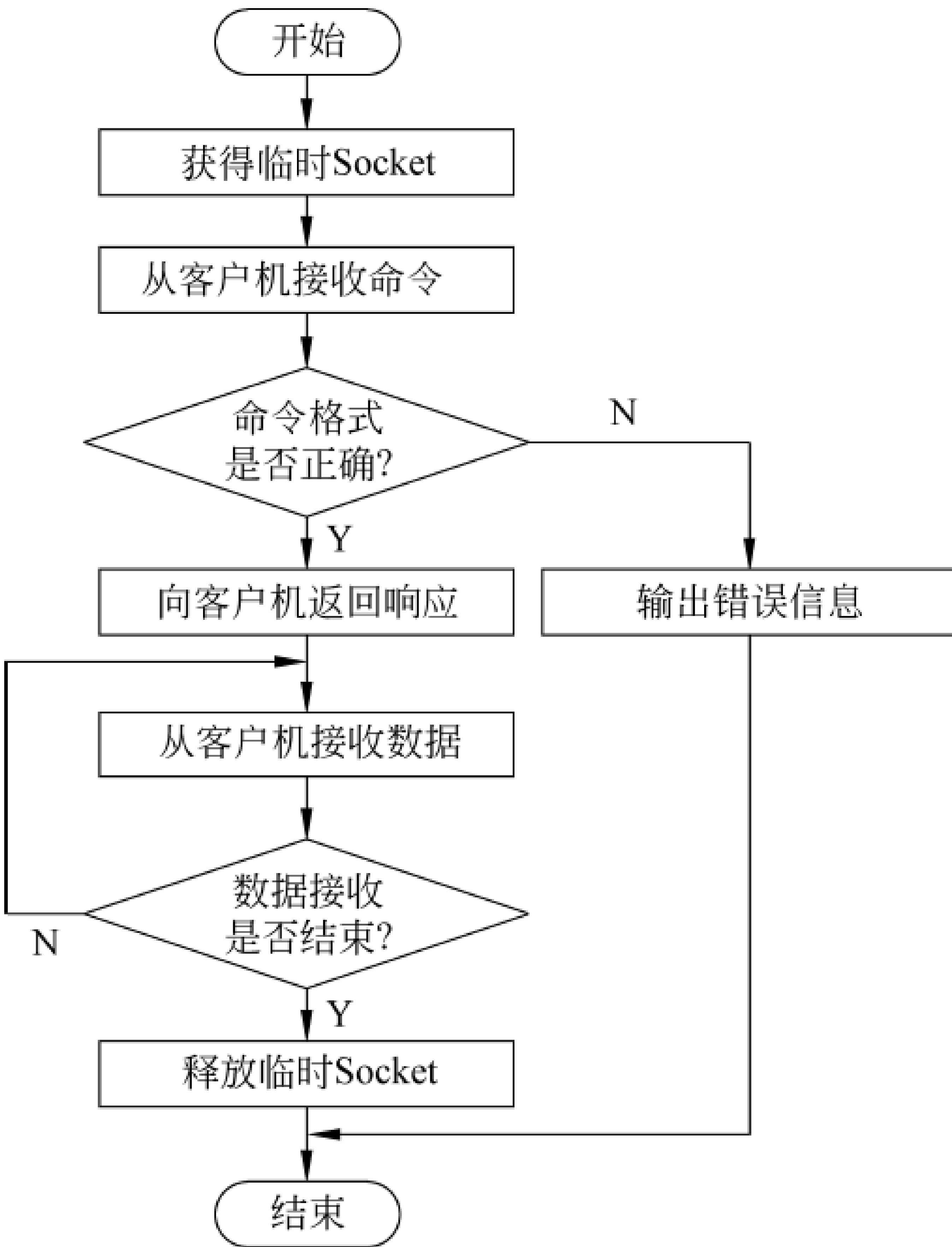


图 12-7　子线程流程图

12.3.3　程序源代码

下面给出基于 TCP 的服务器程序的源代码：

```
//TcpServer.cpp：定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "winsock2.h"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib

struct ThreadParam                               //线程参数数据结构
```

```
{
    SOCKET sock;
    sockaddr_in addr;
};
long ThreadCount=0;
long * aa=&ThreadCount;

DWORD WINAPI ServerThread(LPVOID lpParam)          //客户服务线程
{
    InterlockedIncrement(aa);                      //线程数目加1
    cout<<endl<<"Server 与 Client 建立连接";

    SOCKET tempsock=((ThreadParam * )lpParam)->sock;
    sockaddr_in tempaddr=((ThreadParam * )lpParam)->addr;

    char recvbuf[65535];                           //设置接收缓冲区
    memset(recvbuf,'\0',sizeof(recvbuf));
    int nRecv;
    nRecv=recv(tempsock,recvbuf,sizeof(recvbuf),0);
    if(nRecv==SOCKET_ERROR)                        //通过端口接收数据
    {
        cout<<endl<<"Socket Recv 失败!"<<endl;
        return 0;
    }
    if(strcmp(recvbuf,"sendfile")!=0)              //是否 send file 命令
        return 0;

    int nSend;
    nSend=send(tempsock,"command ok",sizeof("command ok"),0);
    if(nSend==SOCKET_ERROR)                        //通过端口发送数据
    {
        cout<<endl<<"Socket Send 失败!"<<endl;
        return 0;
    }

    nRecv=recv(tempsock,recvbuf,sizeof(recvbuf),0);
    if(nRecv==SOCKET_ERROR)                        //通过端口接收数据
    {
        cout<<endl<<"Socket Recv 失败!"<<endl;
        return 0;
    }
    cout<<endl<<"Server 接收数据:"<<recvbuf<<endl;

    closesocket(tempsock);                         //关闭临时 Socket
```

```
        InterlockedDecrement(aa);                          //线程数目减 1
        cout<<endl<<"Server 接收数据完成"<<endl;
        return 0;
    }

    void main(int argc,char * argv[])
    {
        if(argc!=2)                                        //检查命令行参数
        {
            cout<<endl<<"请按以下格式输入命令行:TcpServer server_port"<<endl;
            return;
        }

        WSADATA WSAData;
        if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)          //套接字异步启动
        {
            cout<<endl<<"WSAStartup 初始化失败"<<endl;
            return;
        }

        SOCKET sock;
        sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);      //创建流式 Socket
        if(sock==INVALID_SOCKET)
        {
            cout<<endl<<"创建 Socket 失败!"<<endl;
            return;
        }

        sockaddr_in serveraddr;                            //初始化本地 Socket
        serveraddr.sin_family=AF_INET;
        serveraddr.sin_port=htons((unsigned short)atoi(argv[1]));
        serveraddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);

        int nBind;
        nBind=bind(sock,(sockaddr *)&serveraddr,sizeof(serveraddr));
        if(nBind==SOCKET_ERROR)                            //端口与 IP 地址绑定
        {
            cout<<endl<<"Socket Bind 失败!"<<endl;
            return;
        }

        int nListen;
```

```
        nListen=listen(sock,SOMAXCONN);                    //在端口上侦听连接
        if(nListen==SOCKET_ERROR)
        {
            cout<<endl<<"Socket Listen 失败!"<<endl;
            return;
        }
        cout<<endl<<"Server 开始侦听"<<atoi(argv[1])<<"端口"<<endl;

        SOCKET tempsock;
        sockaddr_in tempaddr;
        while(true)
        {
            int templen=sizeof(tempaddr);
            tempsock=accept(sock,(sockaddr*)&tempaddr,&templen);
            if(tempsock==INVALID_SOCKET)                  //接受客户机连接
            {
                cout<<endl<<"Socket Accept 失败!"<<endl;
                return;
            }

            if(ThreadCount>=10)                           //线程数是否达到上限
            {
                closesocket(tempsock);                    //关闭临时 Socket
                continue;
            }

            ThreadParam Param;                            //设置传递给线程参数
            Param.sock=tempsock;
            Param.addr=tempaddr;
            DWORD dwThreadId;                             //为客户创建服务线程
            CreateThread(NULL,0,ServerThread,&Param,0,&dwThreadId);
        }

        closesocket(sock);                                //关闭流式 Socket
        WSACleanup();                                     //套接字异步关闭
    }
```

图12-8给出了服务器的执行过程。程序命令行输入为 TcpServer 8000。服务器在指定端口(8000)上侦听客户机的连接请求,与客户机建立相应的TCP连接,然后接收客户机发送的命令与数据内容,并且将接收到的数据显示在控制台上。

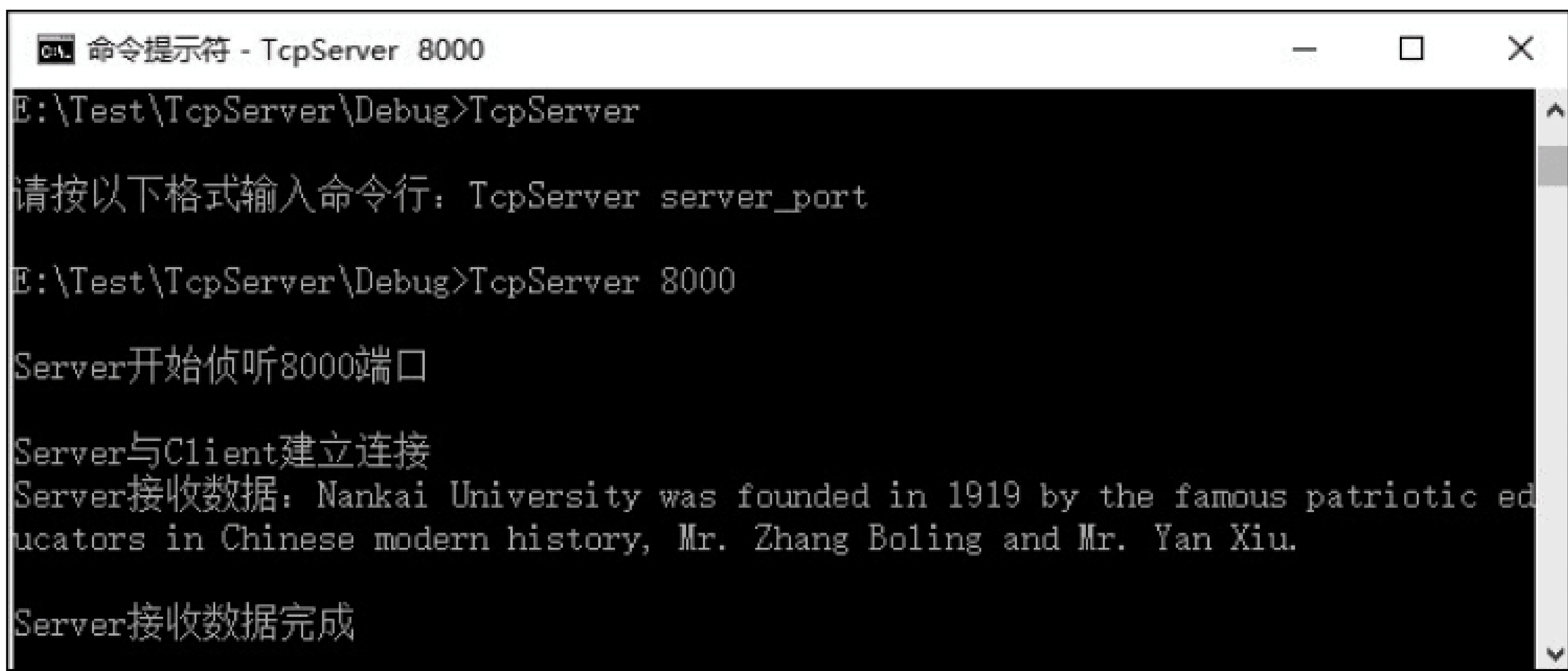


图 12-8　服务器的执行过程

12.4　练　习　题

根据基于 TCP 的客户机/服务器工作模式，编写客户机程序向服务器发送命令，并且根据服务器返回的响应发送数据。客户机向服务器发送 sendfile 命令，服务器向客户机返回 command ok 响应，客户机向服务器发送相应的数据。该数据需要从指定的输入文件中获得。程序设计的具体要求如下。

（1）要求程序为命令行程序。例如，可执行文件名为 TcpClient. exe，则程序的命令行格式为：

```
TcpClient server_addr server_port input_file
```

其中，server_addr 为服务器的 IP 地址，server_port 为服务器的 TCP 端口号，input_file 为保存发送数据的文件。

（2）要求将客户机的状态显示在控制台上，具体格式为：

```
TCP Client 请求与 TCP Server 建立连接
TCP Client 发送数据：…
```

（3）要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限，但是在提交的说明文档中需要加以注明。

（4）要求撰写说明文档，包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第13章 基于UDP的客户机/服务器程序

13.1 设计目的

网络服务是以客户机/服务器模式工作的，服务器在某些特定端口上提供网络服务，等待客户机发送服务请求并进行响应。UDP是不需要建立连接的传输层协议。本章练习的目的是，通过基于UDP的客户机与服务器程序设计，了解UDP协议的基本概念与主要功能，掌握这类网络应用的设计思路与编程方法。

13.2 相关知识

本章涉及的相关知识包括UDP协议的概念与UDP数据包结构。

13.2.1 UDP协议的基本概念

UDP是一种无连接、不可靠的传输层协议。从应用层的角度来看，UDP协议在网络层IP协议的基础上，为应用层的程序提供不可靠的数据包传输服务。图13-1给出了UDP与其他协议的关系。也就是说，UDP为上面的应用层协议提供传输服务。UDP主要用于对传输效率要求高的应用层协议。例如，引导协议(BOOTP)、网络时间协议(NTP)、简单网络管理协议(SNMP)、简单文件传输协议(TFTP)等。另外，域名系统(DNS)可以依赖于TCP或UDP。UDP的主要特点可归纳为无连接、低可靠性、数据报传输。

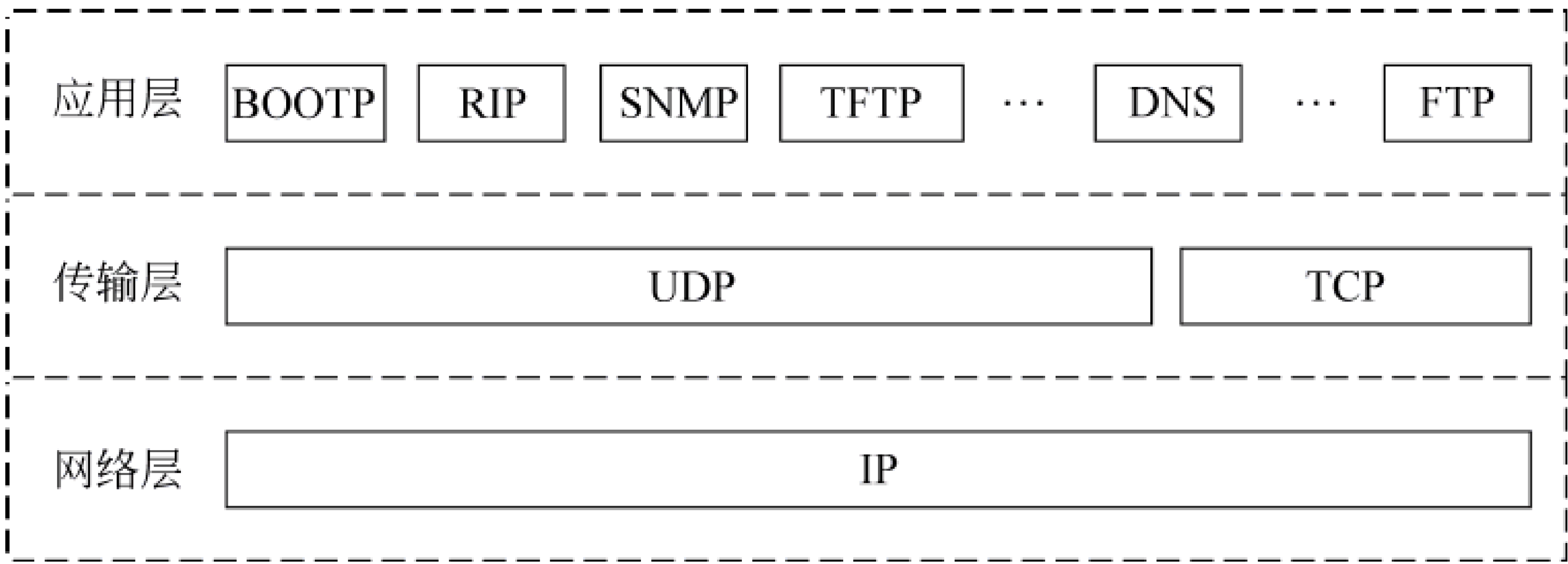


图13-1 UDP与其他协议的关系

UDP协议规定在进行数据传输之前，不需要在通信双方之间建立连接，因此有效地减

少了协议的开销与传输延迟。UDP 除了提供一种可选的校验和之外,几乎没提供其他保证传输可靠性的措施。如果 UDP 协议检测出接收到的数据包出错,它就会丢弃这个出错的数据包,既不确认,也不会要求发送端重新传输。UDP 没有采用基于窗口的流量控制机制,当数据包过多时在接收端有可能出现溢出,同样既不确认,也不要求发送端重新传输。因此,UDP 协议提供的是"尽力而为"的传输服务。

UDP 协议对于应用程序提交的高层数据,在添加 UDP 头部形成 UDP 数据包后,向下提交给网络层的 IP 协议来处理。在发送端,UDP 对应用层数据既不合并,也不拆分,而是保留数据原来的长度与格式。在接收端,UDP 将接收的数据包原封不动地提交给应用程序。因此,在使用 UDP 协议时,应用程序必须选择长度合适的高层数据。如果应用程序提交的数据太短,则协议开销相对较大;如果应用程序提交的数据太长,UDP 向网络层提交的 UDP 数据包可能被分片,这样也会降低协议的效率。

应用程序选择是否采用 UDP 协议时,存在以下几个需要考虑的原则。

(1) 简短的交互式应用:如果客户机与服务器之间只需要交互简短的请求与应答,应用程序在这种情况下应该选择 UDP 协议。应用程序可以通过设置定时器与重传机制,以便处理网络层的 IP 分组丢失问题。

(2) 视频播放应用:在互联网环境下播放视频,用户最关注的是视频流能尽快地、不间断地播放,丢失个别数据包对播放效果不会产生重要影响。如果采用 TCP 协议,可能因重传丢失数据包而增大传输延迟,反而对视频播放造成不利的影响。因此,视频播放应用对数据交付实时性要求较高,而对数据交付可靠性要求较低,UDP 协议显然更为适用。

(3) 多播与广播应用:对于 IP 电话、视频会议等实时性应用,它们要求源主机以恒定速率发送数据,在拥塞发生时允许丢弃部分数据包。UDP 协议支持一对一、一对多与多对多的交互式通信,这点也是 TCP 协议所不支持的。因此,对于多播与广播类应用,UDP 协议显然更为适用。

13.2.2 UDP 数据包的结构

根据 OSI 参考模型的定义,传输层使用下面的网络层提供的服务,并且要向上面的应用层提供服务。由于 UDP 协议所处的层次高于 IP 协议,因此 UDP 数据包需要封装在 IP 分组中传输。图 13-2 给出了 UDP 数据包的封装过程。当某种应用进程是基于 UDP 协议时,首先将应用层数据作为 UDP 数据与 UDP 头部封装成 UDP 包,然后将 UDP 包作为 IP 数据与 IP 头部封装成 IP 分组。在将 UDP 数据包封装成 IP 分组时,IP 头部的协议字段表示上层协议类型,用于表示 UDP 协议的字段值为 17。

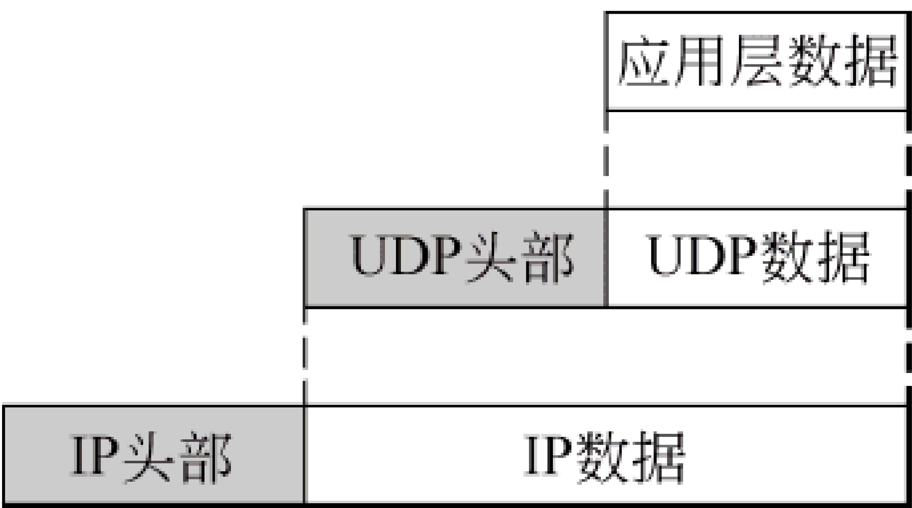


图 13-2 UDP 数据包的封装过程

设计 UDP 协议的主要原则是协议简单、运行快捷。RFC768 是最早出现的 UDP 协议文档,它描述了 UDP 协议的基本内容。RFC1122 是对 UDP 协议进行补充的 RFC 文档。UDP 提供端口形式的传输层寻址,以及一种可选的校验和功能。UDP 协议制定了统一的 UDP 数据包格式。UDP 数据包分为两个部分:UDP 头部与 UDP 数据。图 13-3 给出了 UDP 数据包的结构。UDP 头部长度固定为 8 字节,UDP 数据部分的长度是可变的。



图 13-3　UDP 数据包的结构

UDP 头部由以下这些字段组成。

1. 端口号

端口号(port number)字段包括两个部分：源端口号与目的端口号。源端口号与目的端口号字段的长度均为 16 位。源端口号表示源进程(发送端)使用的 UDP 端口号,目的端口号表示目的进程(接收端)使用的 UDP 端口号。如果源进程是客户机,源端口号是由 UDP 软件分配的临时端口号,目的端口号使用服务器的熟知端口号。

2. UDP 长度

UDP 长度(length)字段的长度为 16 位,表示包括 UDP 头部在内的 UDP 数据包的总长度。UDP 数据包的最小长度为 8B,最大长度为 65 535B。由于 UDP 头部长度固定为 8B,因此 UDP 数据部分的最大长度为 65 527B。

3. UDP 校验和

UDP 校验和(checksum)字段的长度为 16 位,用来检查 UDP 数据包在传输中是否出错,其计算方法为 IP 头部校验和的计算方法相同。UDP 校验和字段的校验范围为伪头部、UDP 头部与 UDP 数据。如果应用程序对通信效率的要求高于可靠性,它可以选择不使用 UDP 校验和,这样设计反映出效率优先的思想。

伪头部(pseudo header)本身不是 UDP 数据包的真正头部,只是在计算校验和时临时和 UDP 数据包相加。伪头部的长度为 12B。图 13-4 给出了伪头部的结构。伪头部内容主要来自 IP 分组头部的一部分,它包括以下几个字段：源 IP 地址(16 位)、目的 IP 地址(16 位)、保留位(8 位)、协议(8 位)与 UDP 长度(16 位)。为了保证头部长度为 16 位的整数倍,伪头部中还有 8 位的填充部分(全 0)。UDP 长度是 UDP 数据包的长度,不包括伪头部的长度。



图 13-4　伪头部的结构

13.2.3　基于 UDP 的客户机/服务器编程

基于 UDP 的网络应用采用客户机/服务器模式。这里,客户机与服务器是互相通信的两个应用程序的进程,它们分别称为客户机与服务器。图 13-5 给出了基于 UDP 的客户机/服务器结构。客户机是使用某种网络服务的应用进程,服务器是提供某种网络服务的应用

进程。在客户机中,建立一个输出队列与一个输入队列,它们分别用于发送请求与接收应答。在服务器中,建立一个输入队列与一个输出队列,它们分别用于接收请求与发送应答。每种队列都采用先到先服务的工作模式。这种服务器工作方式被称为重复服务器。

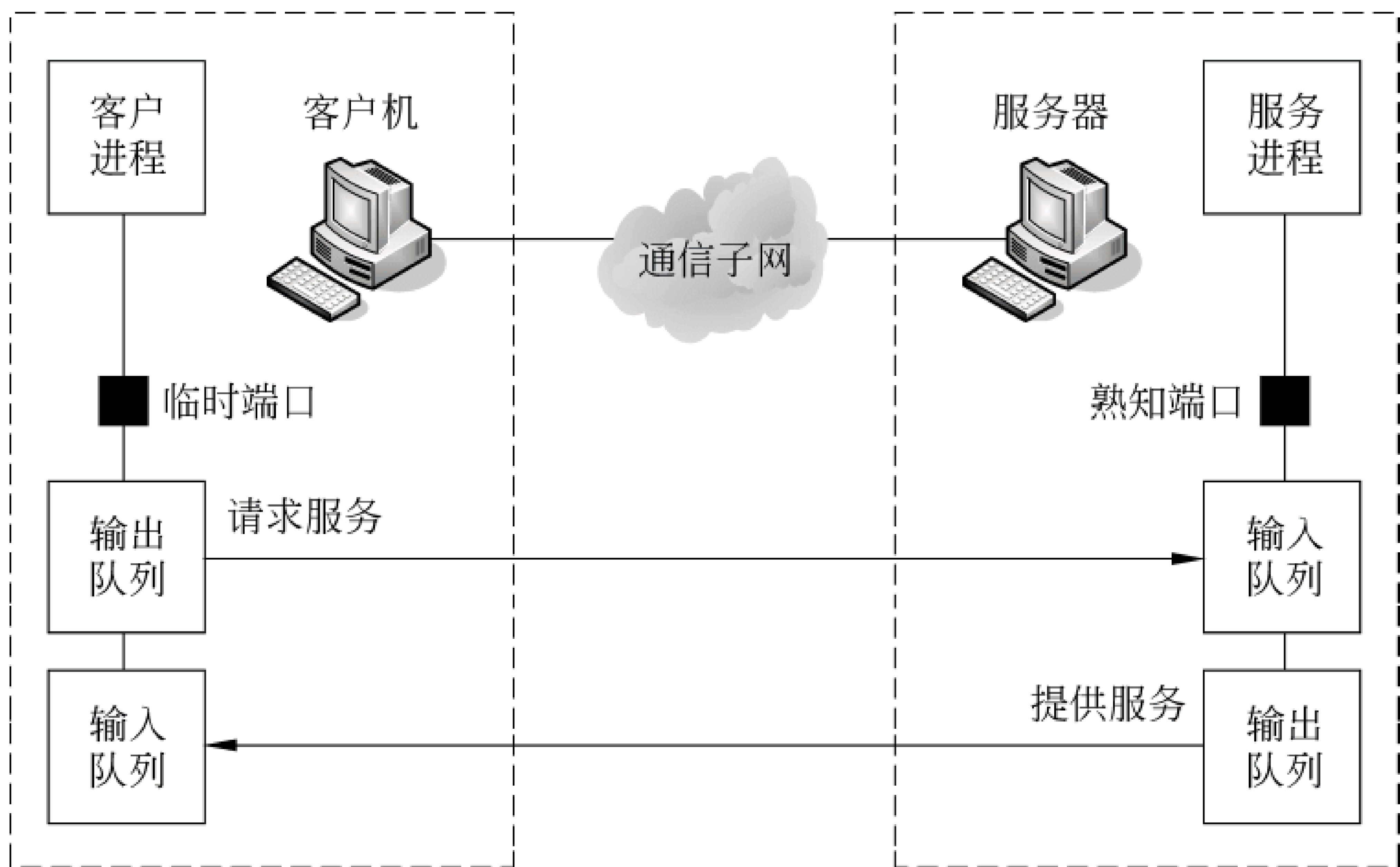


图 13-5　基于 UDP 的客户机/服务器结构

无论是客户机还是服务器进程,它们都要通过 IP 地址与端口号来加以标识。在基于 UDP 的网络应用中,使用的端口号是 UDP 协议的端口号。客户机是使用网络服务的应用进程,它通过临时端口号向服务器请求服务。服务器是提供网络服务的应用进程,为了使众多的客户机知道服务器的存在,它通过熟知端口号来向客户机提供服务。表 13-1 给出了 UDP 的主要熟知端口号。这种熟知端口号(0～1023)是由 IANA 来统一分配的,每个客户机都知道相应服务器的熟知端口号。

表 13-1　UDP 的主要熟知端口号

端　口　号	服 务 进 程	说　　明
53	DNS	域名系统
67	BOOPS	引导协议(服务器)
68	BOOPC	引导协议(客户机)
69	TFTP	简单文件传输协议
111	RPC	远程过程调用
123	NTP	网络时间协议
161	SNMP	简单网络管理协议
162	SNMP	简单网络管理协议(Trap)

13.3　例 题 分 析

13.3.1　设计要求

根据基于 UDP 的客户机/服务器工作模式,编写服务器程序接收客户机的命令,并根据命令向客户机做出响应。当客户机向服务器发送 getfile 命令时,服务器向客户机发送指

定文件中的数据;当客户机向服务器发送 gettime 命令时,服务器向客户机发送当前系统时间。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 UdpServer.exe,则程序的命令行格式为:

```
UdpServer server_port
```

其中,server_port 为服务器侦听的 UDP 端口号。

(2) 要求将服务器的状态显示在控制台上,具体格式为:

```
UDP Server 接收命令:…
UDP Server 发送数据:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

13.3.2 关键问题

1. 基本编程模式分析

基于 UDP 的客户机/服务器进程有相对固定的编程模式。如果客户机与服务器进程之间通信,需要依次调用 Socket 提供的不同函数来实现。但是,服务器编程比客户机编程更为复杂。服务器采用重复服务器方式处理多个服务请求。图 13-6 给出了基于 UDP 的客户机/服务器编程模式。对于客户机与服务器进程,它们首先需要调用 socket()函数建立套接字,然后可以调用 sendto()函数发送数据,或者调用 recvfrom()函数接收数据,最后需要调用 closesocket()函数关闭套接字。但是,服务器进程在发送与接收数据之前,还要调用

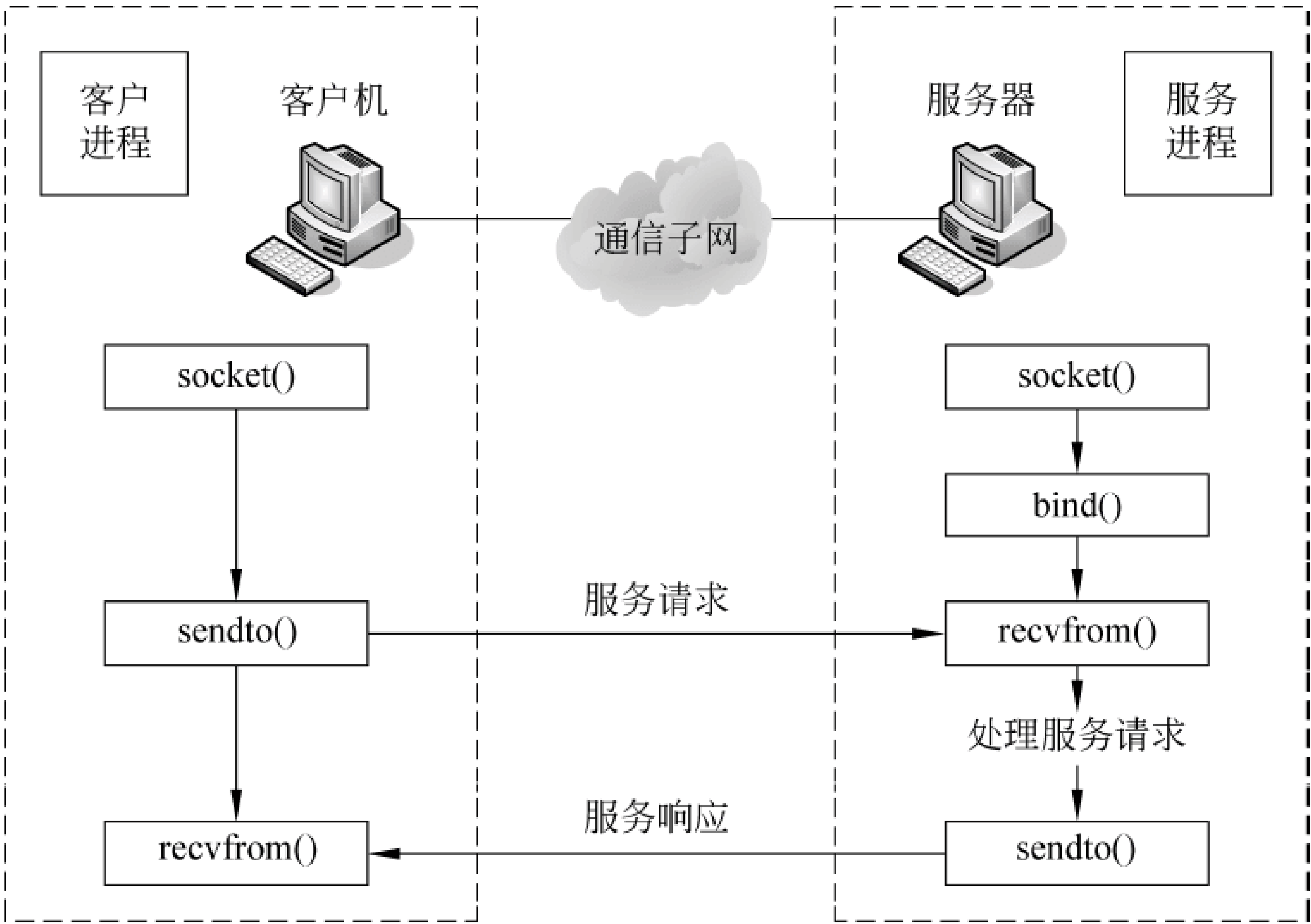


图 13-6　基于 UDP 的客户机/服务器编程模式

bind()函数将某个端口与套接字绑定。

2. 创建数据报套接字

为了实现基于 UDP 的客户机/服务器进程，首先需要调用 socket()函数创建套接字，其中的 SOCK_DGRAM 表示创建数据报套接字，IPPROTO_IP 表示采用 IP 协议。接着，需要使用 INADDR_ANY 来获得本地主机的 IP 地址，然后将 IP 地址与端口号共同填充本地 Socket 结构。最后，调用 bind()函数将某个端口与套接字绑定。

下面给出创建数据报套接字的伪代码：

```
//创建数据报 Socket
socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
//填充本地 Socket 地址
sockaddr_in serveraddr;
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons((unsigned short)atoi(argv[1]));
serveraddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
//将端口与 IP 地址绑定
bind(sock,(sockaddr*)&serveraddr,sizeof(serveraddr));
```

3. 在主循环中发送与接收数据

服务器需要处理多个客户机的服务请求，因此需要采用重复服务器工作方式。服务器在主循环中调用 recvfrom()函数接收客户机的请求，然后根据服务请求来做出相应的处理。当客户机向服务器发送 getfile 命令时，服务器调用 sendto()函数向客户机发送文件 input 中的数据；当客户机向服务器发送 gettime 命令时，服务器调用 sendto()函数向客户机发送当前系统时间。最后，需要调用 closesocket()函数释放数据报套接字。注意，服务器在发送与接收数据之前，首先需要初始化发送与接收缓冲区。

下面给出在线程中发送与接收数据的伪代码：

```
//在主循环中发送与接收数据
while(true)
{
    //初始化接收缓冲区
    char recvbuf[20];
    memset(recvbuf,'\0',sizeof(recvbuf));
    //从客户机接收请求
    nRecv=recvfrom(sock,recvbuf,sizeof(recvbuf),0,(sockaddr*)&clientaddr,
    &clientaddrlen);
    //初始化发送缓冲区
    char sendbuf[1500];
    memset(sendbuf,'\0',sizeof(sendbuf));
    //判断客户机请求的类型
    if(strcmp(recvbuf,"getfile")==0)
    {
        //将文件数据写入发送缓冲区
```

```
            fstream infile;
            infile.open("input",ios::in|ios::nocreate);
            infile.read(sendbuf,nlength);
            //向客户机发送数据
            nSend=sendto(sock,sendbuf,sizeof(sendbuf),0,(sockaddr * )&clientaddr,
            clientaddrlen);
        }
        //判断客户机请求的类型
        if(strcmp(recvbuf,"gettime")==0)
        {
            //将系统时间写入发送缓冲区
            time_t CurTime;
            time(&CurTime);
            strftime(sendbuf,sizeof(sendbuf),"%Y-%m-%d %H:%M:%S",localtime(&CurTime));
            //向客户机发送数据
            nSend=sendto(sock,sendbuf,sizeof(sendbuf),0,(sockaddr * )&clientaddr,
            clientaddrlen);
        }
    }
```

4. 程序流程图

图13-7给出了主程序流程图。要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要提供一个服务器端口号。如果命令行参数的个数不是一个,则程序在输出错误信息后退出。在主程序的流程中,需要判断命令格式是否正确,以及数据发送是否结束。

13.3.3 程序源代码

下面给出基于UDP的服务器程序的源代码:

```
//UdpServer.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "time.h"
#include "winsock2.h"
#include "fstream"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib

void main(int argc,char *  argv[])
```

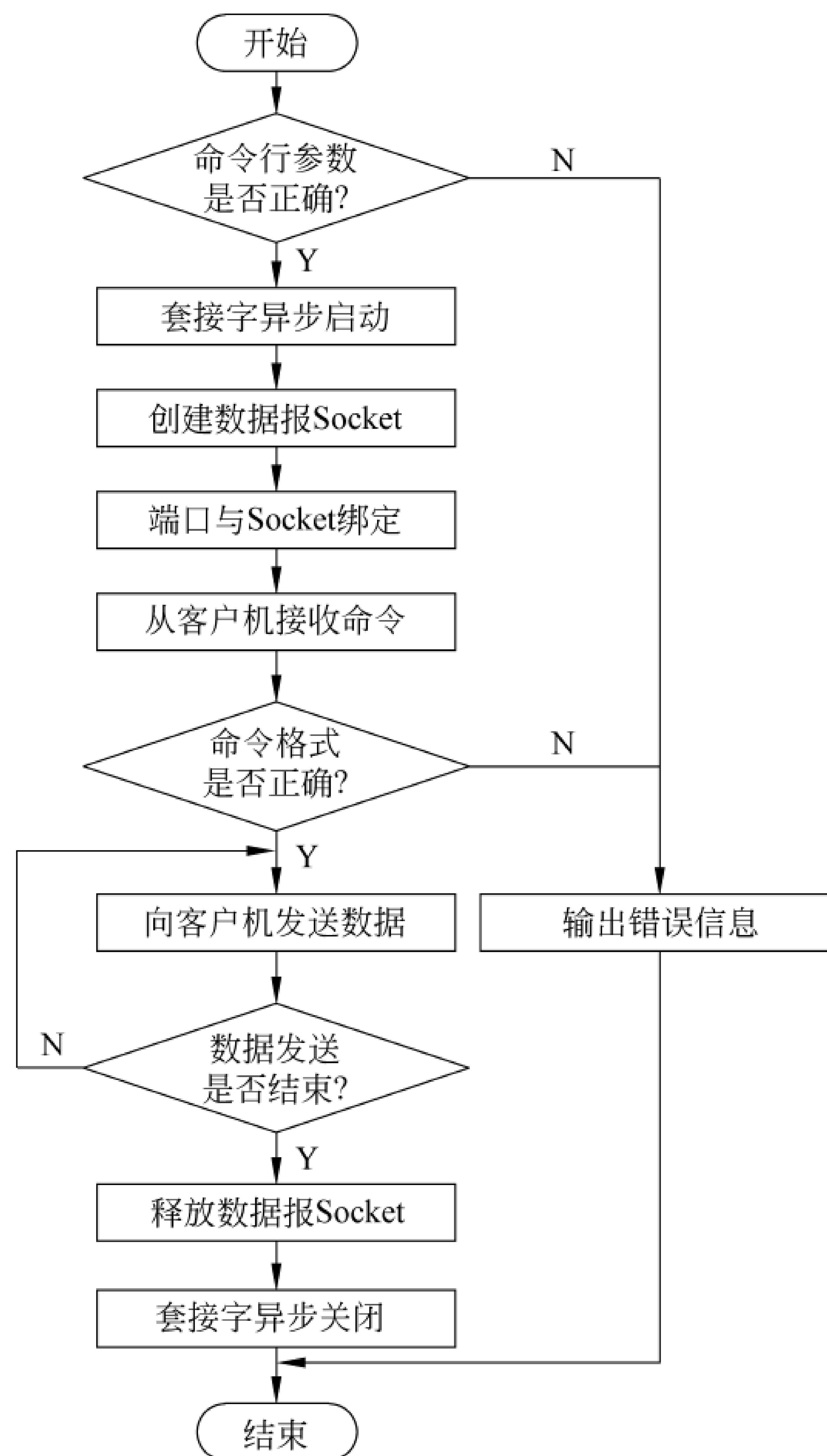


图 13-7　主程序流程图

```
{
    if(argc!=2)                                     //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:UdpServer server_port"<<endl;
        return;
    }

    WSADATA WSAData;
    if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)       //套接字异步启动
    {
        cout<<endl<<"WSAStartup 初始化失败"<<endl;
        return;
    }

    SOCKET sock;                                    //创建数据报 Socket
```

```
sock=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if(sock==INVALID_SOCKET)
{
    cout<<endl<<"创建 Socket 失败!"<<endl;
    return;
}

sockaddr_in serveraddr;                          //初始化本地 Socket
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons((unsigned short)atoi(argv[1]));
serveraddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
int serveraddrlen=sizeof(serveraddr);

int nBind;
nBind=bind(sock,(sockaddr*)&serveraddr,sizeof(serveraddr));
if(nBind==SOCKET_ERROR)                          //端口与 IP 地址绑定
{
    cout<<endl<<"Socket Bind 失败!"<<endl;
    return;
}

sockaddr_in clientaddr;                          //初始化远程 Socket
int clientaddrlen=sizeof(clientaddr);

while(true)
{
    char recvbuf[20];                            //设置接收缓冲区
    memset(recvbuf,'\0',sizeof(recvbuf));

    int nRecv;
    nRecv= recvfrom(sock, recvbuf, sizeof(recvbuf), 0, (sockaddr*) &clientaddr,
    &clientaddrlen);
    if(nRecv==SOCKET_ERROR)                      //通过端口接收数据
    {
        cout<<endl<<"Socket Recv 失败!"<<endl;
        return;
    }
    cout<<endl<<"UDP Server 接收命令:"<<recvbuf<<endl;

    char sendbuf[1500];                          //设置发送缓冲区
    memset(sendbuf,'\0',sizeof(sendbuf));

    if(strcmp(recvbuf,"getfile")==0)             //是否为 getfile 命令
    {
```

```
        fstream infile;                         //打开输入文件
        infile.open("input",ios::in);
        infile.seekg(0,ios::end);
        int nlength=infile.tellg();             //获得输入文件长度
        infile.seekg(0,ios::beg);
        infile.read(sendbuf,nlength);           //数据读入 sendbuf

        int nSend;
        nSend=sendto(sock,sendbuf,strlen(sendbuf),0,(sockaddr*)
        &clientaddr,clientaddrlen);
        if(nSend==SOCKET_ERROR)                 //通过端口发送数据
        {
            cout<<endl<<"Socket Send 失败!"<<endl;
            return;
        }
    }

    if(strcmp(recvbuf,"gettime")==0)            //是否为 gettime 命令
    {
        time_t CurTime;
        time(&CurTime);                         //获得当前系统时间
        strftime(sendbuf,sizeof(sendbuf),"%Y-%m-%d %H:%M:%S",
        localtime(&CurTime));

        int nSend;
        nSend=sendto(sock,sendbuf,sizeof(sendbuf),0,(sockaddr*)
        &clientaddr,clientaddrlen);
        if(nSend==SOCKET_ERROR)                 //通过端口发送数据
        {
            cout<<endl<<"Socket Send 失败!"<<endl;
            return;
        }
    }
    cout<<"Server 发送数据:"<<sendbuf<<endl;

    nRecv= recvfrom (sock, recvbuf, sizeof (recvbuf), 0, (sockaddr * ) &clientaddr,
    &clientaddrlen);
    if(nRecv==SOCKET_ERROR)                     //通过端口接收数据
    {
        cout<<endl<<"Socket Recv 失败!"<<endl;
        return;
    }
    if(strcmp(recvbuf,"command ok")!=0)
        return;
```

```
        }

        cout<<endl<<"Server 发送数据完成";
        closesocket(sock);                          //关闭数据报 Socket
        WSACleanup();                               //套接字异步关闭
    }
```

图13-8给出了服务器的执行过程。程序命令行输入为UdpServer 8000,服务器在指定端口(8000)上接收客户机发送的命令,然后根据命令将相应数据发送给客户机,并将接收命令与发送数据显示在控制台上。



图13-8 服务器的执行过程

13.4 练 习 题

根据基于UDP的客户机/服务器工作模式,编写客户机程序向服务器发送命令,并根据服务器发送的数据做出响应。当客户机向服务器发送getfile命令时,服务器向客户机发送指定文件中的数据;当客户机向服务器发送gettime命令时,服务器向客户机发送当前系统时间。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为UdpClient.exe,则程序的命令行格式为:

```
UdpClient server_addr server_port command
```

其中,server_addr为服务器的IP地址,server_port为服务器的UDP端口号,command为客户机发送的命令。

(2) 要求将服务器的状态显示在控制台上,具体格式为:

```
UDP Client 发送命令:…
UDP Client 接收数据:…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 14 章 FTP 客户机程序设计

14.1 设计目的

Internet 提供了很多类型的网络服务，这些服务实际上都是应用层的服务。网络服务是以客户机/服务器模式工作的。FTP 服务是基于 TCP 协议的网络服务。本章练习的目的是，通过 FTP 客户机程序的设计，了解 FTP 服务的基本概念与主要功能，掌握应用层服务的设计思路与编程方法。

14.2 相关知识

本章涉及的相关知识包括应用层的概念、FTP 服务的概念与工作原理，以及 FTP 命令与应答格式。

14.2.1 应用层的基本概念

应用层是网络体系结构的最高层次。无论是 OSI 参考模型还是 TCP/IP 参考模型，它们的最高层次都是应用层。图 14-1 给出了 OSI 与 TCP/IP 两种参考模型的层次结构。应用层提供了各种类型的网络服务，Internet 技术的发展极大地丰富了应用层的内容。每种应用层服务都有对应的协议标准。目前，应用层协议主要包括以下几种：文件传输协议(FTP)、远程登录(Telnet)、简单邮件传输协议(SMTP)、超文本传输协议(HTTP)、域名系统(DNS)与简单网络管理协议(SNMP)等。

OSI参考模型	TCP/IP参考模型
应用层	应用层
表示层	
会话层	
传输层	互联层
网络层	网络层
数据链路层	主机-网络层
物理层	

图 14-1　OSI 与 TCP/IP 两种参考模型的层次结构

两种参考模型都是从上到下存在着单项依赖关系，应用层协议需要直接使用传输层提供的服务，而传输层又需要通过网络层的 IP 协议来收发数据。按照应用层协议与传输层服务的依赖关系，应用层协议可以分为三种类型：①只依赖于 TCP 的应用层协议，例如 FTP、Telnet、SMTP 与 HTTP 等；②只依赖于 UDP 的应用层协议，例如 SNMP、TFTP 等；③可依赖于 TCP 或 UDP 的应用层协议，例如 DNS。

14.2.2　FTP 服务的基本概念

文件传输服务又称为 FTP 服务，这是因为它遵循 TCP/IP 协议族中的文件传输协议(File Transfer Protocol，FTP)。FTP 服务允许用户将文件从一台计算机传输到另一台计算机，并保证文件在 Internet 中传输的可靠性。Internet 采用 TCP/IP 协议作为基本协议，无论两台计算机在地理位置上相距多远，只要这两台计算机都支持 FTP 协议，则它们之间都可以相互传输文件。

用户使用 FTP 服务，首先要登录 FTP 服务器，这时就要知道 FTP 服务器名或 IP 地址。每个 FTP 服务器都有自己的 FTP 服务器名，这个名字在全球范围内是唯一的。例如，南开大学 FTP 服务器名为 ftp.nankai.edu.cn。其中，ftp 表示 FTP 服务，nankai.edu.cn 表示南开大学的主机。当用户要登录到某个 FTP 服务器时，还需要输入用户名与密码。Internet 中的一些 FTP 站点是专用的，只允许拥有合法账号的用户进入。

目前，很多 FTP 服务器都提供匿名 FTP 服务，这类 FTP 服务器称为匿名 FTP。提供匿名 FTP 服务的实质是：FTP 服务器中有公开的用户名(一般为 anonymous)，并赋予该用户访问公共目录的权限。如果用户要访问这些匿名 FTP 服务器，可以使用 anonymous 作为用户名，一般不需要输入用户密码。图 14-2 给出了匿名 FTP 服务的例子。为了保证 FTP 服务器的安全，大多数的匿名 FTP 只提供下载服务。

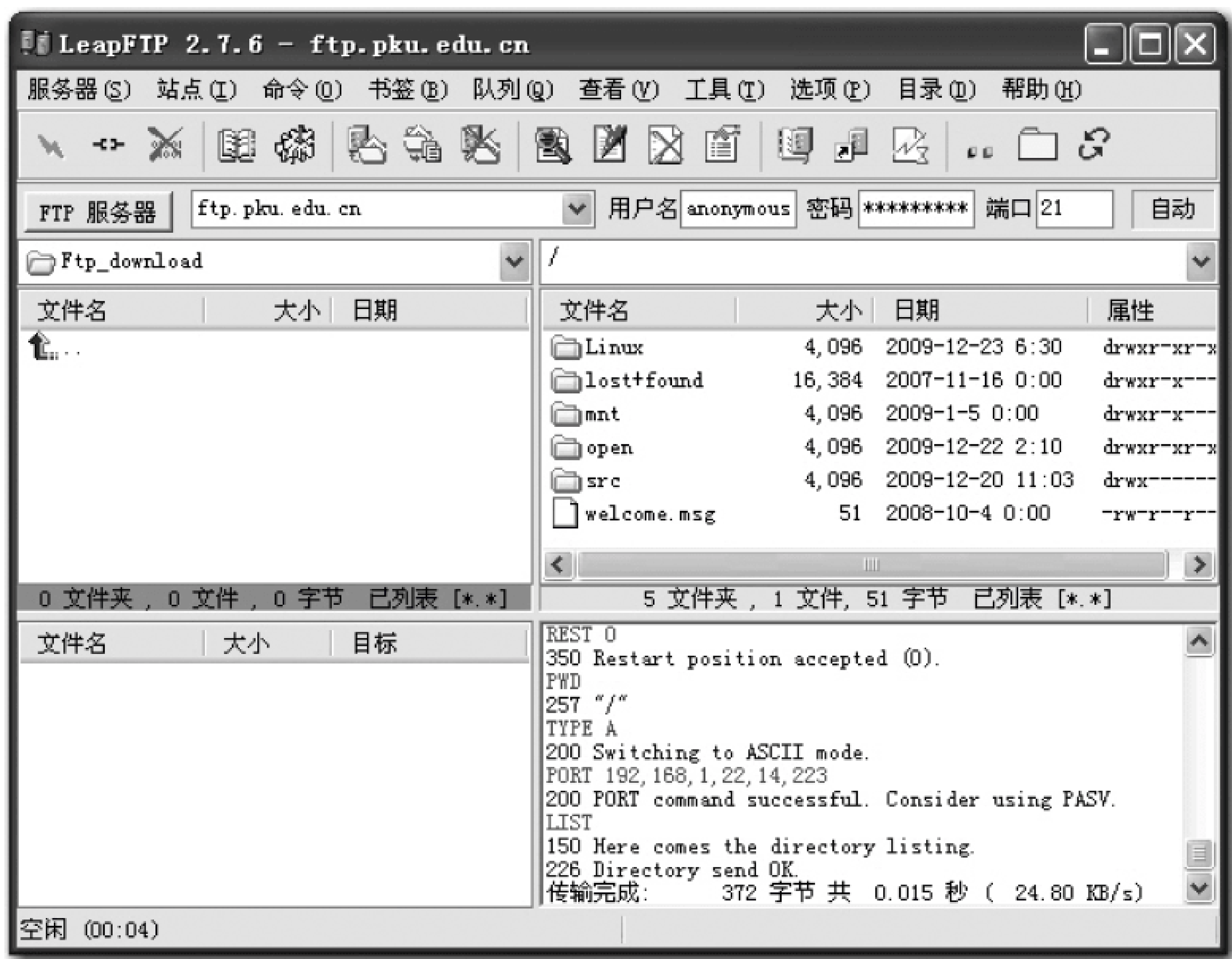


图 14-2　匿名 FTP 服务的例子

FTP 服务与其他 Internet 服务一样，采用的也是客户机/服务器模式。FTP 服务器是指提供 FTP 服务的计算机，其中需要运行 FTP 服务器程序；FTP 客户机是指用户的本地计算机，其中需要运行 FTP 客户端软件。文件在 FTP 服务器中是以目录结构存储的。实际上，FTP 服务器运行着一个 FTP 守护进程(daemon)，它负责为用户提供下载与上载服务。图 14-3 给出了下载与上载的概念。下载是指将文件从 FTP 服务器传输到 FTP 客户机；而

上载是指将文件从 FTP 客户机传输到 FTP 服务器。

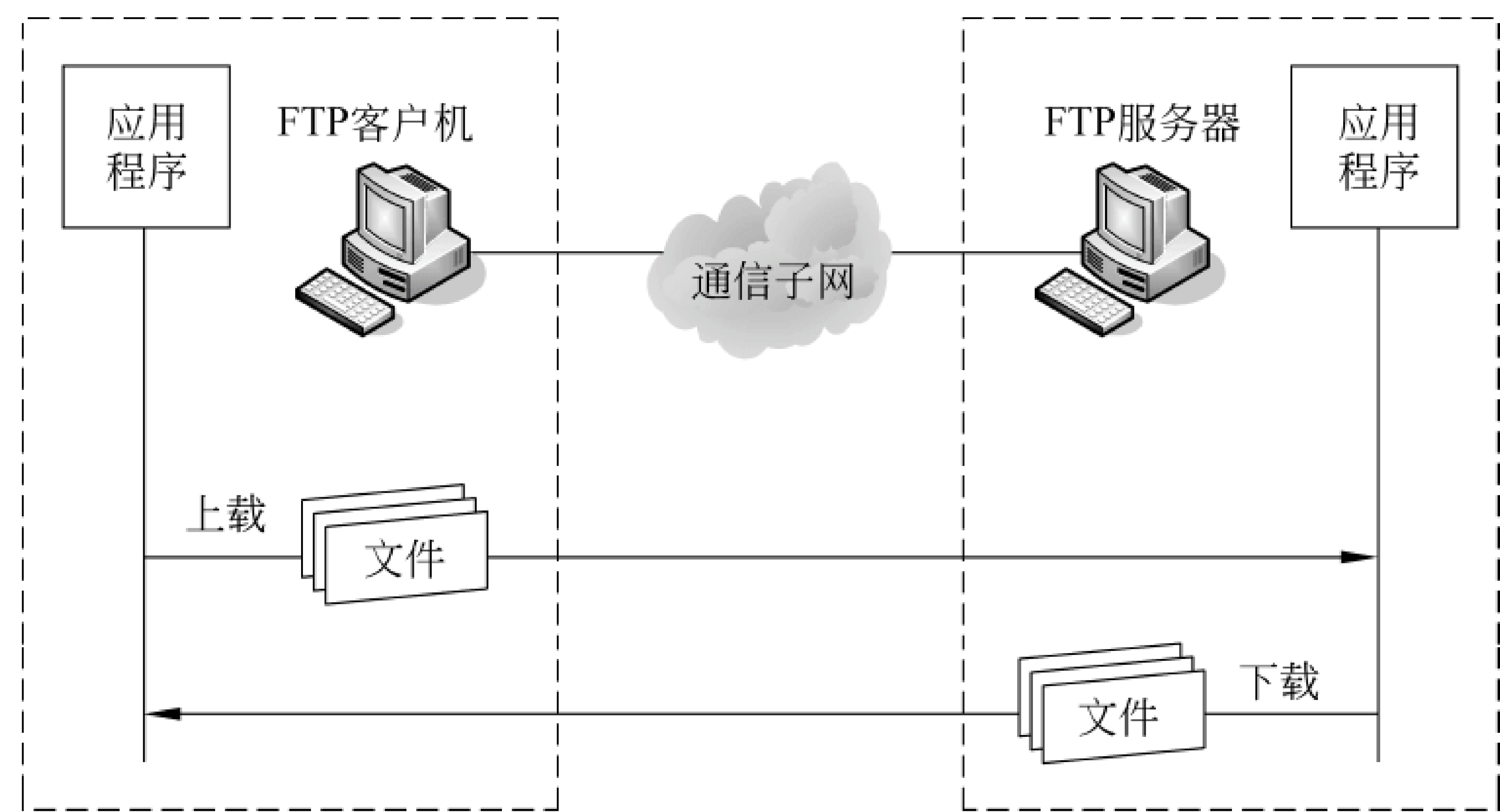


图 14-3　下载与上载的概念

14.2.3　FTP 服务的工作原理

由于 FTP 服务在传输层采用的是 TCP 协议，因此在进行文件传输之前需要先建立连接，要经过建立连接、传输数据与释放连接的基本过程。FTP 服务的特点是数据量大、控制信息相对较少，因此将数据分为控制信息与数据分别处理，这样用于通信的 TCP 连接也相应分为控制连接与数据连接两种。其中，控制连接用于在通信双方之间传输 FTP 命令与应答信息，完成连接建立、身份认证与异常处理等控制操作；数据连接用于在通信双方之间传输文件或目录信息。

图 14-4 给出了 FTP 服务的工作原理。FTP 客户机向 FTP 服务器发送服务请求，FTP 服务器接收与响应 FTP 客户机的请求，并向 FTP 客户机提供所需要的文件传输服务。根据 TCP 协议的规定，FTP 服务器必须使用熟知端口号来提供服务，FTP 客户机使用临时端口号来发送服务请求。FTP 协议为控制连接与数据连接规定了不同的熟知端口号，控制连接使用端口号 21 响应连接建立请求，数据连接使用端口号 20 响应连接建立请求。RFC959 规定了 FTP 协议的工作流程、命令与应答等。

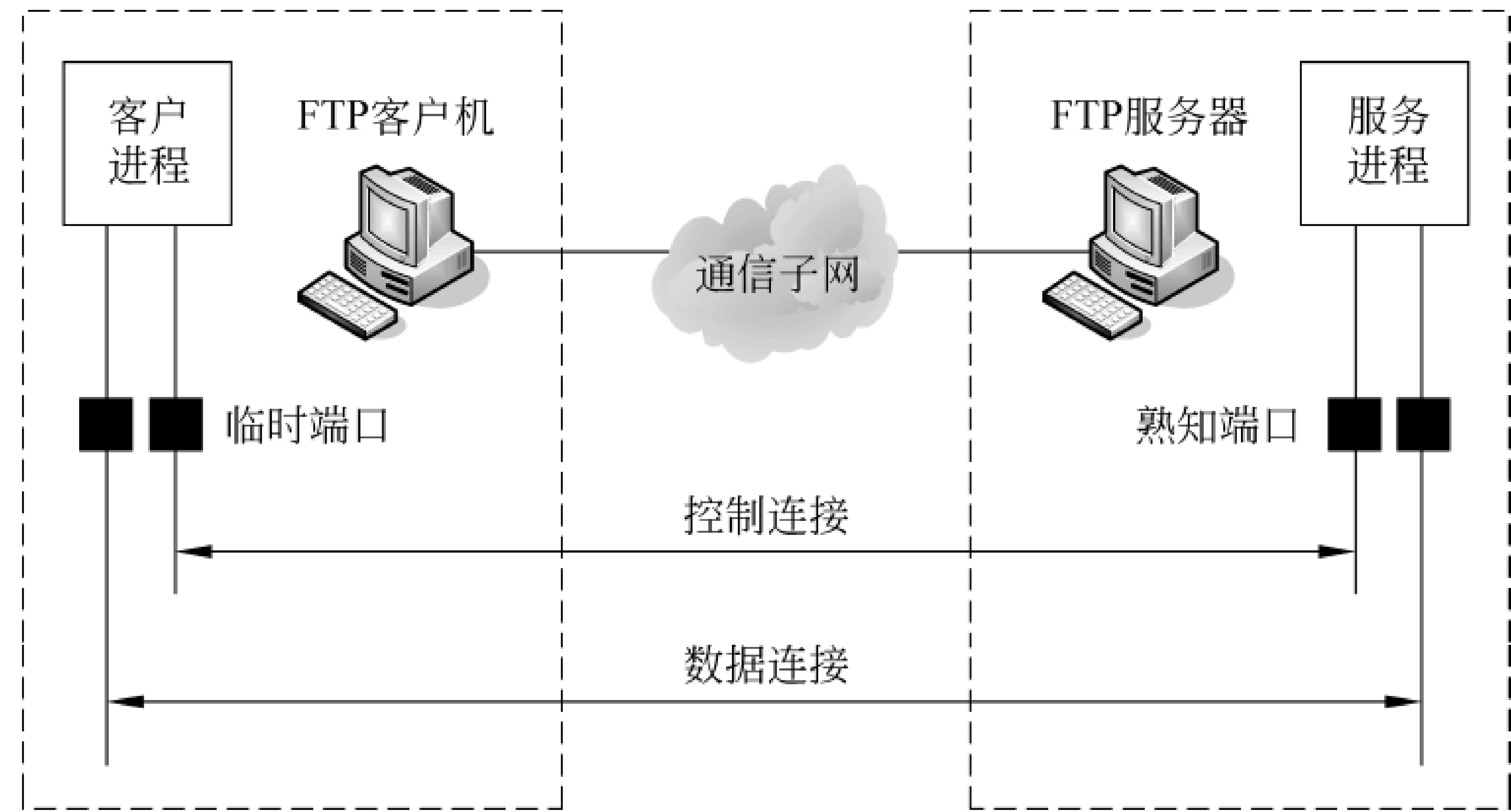


图 14-4　FTP 服务的工作原理

FTP 协议规定了控制连接与数据连接建立和释放的顺序。基本规则是控制连接要在数据连接建立之前建立,控制连接要在数据连接释放之后释放。只有在建立数据连接之后才能传输数据,并且在数据传输过程中需要保持控制连接不中断。控制连接与数据连接的建立与释放也有规定的发起者。控制连接与数据连接建立的发起者只能是 FTP 客户机;控制连接释放的发起者只能是 FTP 客户机,但数据连接释放的发起者可以是 FTP 客户机,也可以是服务器。如果在数据连接保持的情况下控制连接中断,这时可以由 FTP 服务器要求释放数据连接。

FTP 服务的基本工作流程是:FTP 客户机向服务器请求建立控制连接,FTP 客户机与服务器之间会建立一条控制连接;FTP 客户机请求登录到服务器,FTP 服务器要求客户机提供用户名与密码;当 FTP 客户机成功登录到服务器后,FTP 客户机通过控制连接向服务器发出命令,FTP 服务器也通过控制连接向客户机返回响应信息;当 FTP 客户机向服务器发送列出目录命令,FTP 服务器会通过控制连接返回应答信息,并通过新建立的数据连接返回目录信息,这时 FTP 客户机显示的是服务器中的目录结构。

如果用户想改变 FTP 服务器的当前目录,FTP 客户机通过控制连接向服务器发出改变目录命令,FTP 服务器通过数据连接返回改变后的目录列表;如果用户想下载目录中的某个文件,FTP 客户机通过控制连接向服务器发出下载命令,FTP 服务器通过数据连接将文件传输到客户机。数据连接可以通过两种模式打开:ASCII 模式或二进制模式。其中,ASCII 模式适合传输文本文件,二进制模式适合传输二进制文件。数据连接在目录列表或文件下载后关闭,而控制连接在登录结束后才会关闭。

14.2.4 FTP 命令与应答

1. FTP 命令

FTP 协议详细规定了每种协议动作的实现顺序。FTP 命令是 FTP 客户机向服务器发送的操作请求,FTP 服务器根据操作情况向客户机返回应答信息。

FTP 命令的标准书写格式为:

```
命令名 <参数>
```

FTP 命令由两个部分组成:命令名与参数。其中,命令名是由 3 或 4 个大写字母组成的字符串,它是对该命令的英文描述的缩写,例如 USER 命令是 User Name 的缩写;参数是完成命令需要使用的附加信息,例如 USER 命令的参数为用户名。表 14-1 给出了常用的 FTP 命令。FTP 客户机与服务器需要按照上述格式实现 FTP 命令,以保证不同开发者的 FTP 客户机与服务器之间可以交互。

表 14-1 常用的 FTP 命令

FTP 命令名	格　式	用　途
USER	USER <username>	系统登录需要的用户名
PASS	PASS <password>	系统登录需要的密码
LIST	LIST <directory>	列出当前目录中的信息
CWD	CWD <directory>	改变当前目录

续表

FTP 命令名	格　　式	用　　途
MKD	MKD <directory>	在当前目录中创建新目录
RMD	RMD <directory>	删除指定目录
TYPE	TYPE <datatype>	改变文件类型
STOR	STOR <filename>	上载文件到服务器当前目录
RETR	RETR <filename>	下载文件到本地当前目录
DELE	DELE <filename>	删除指定文件
HELP	HELP <command>	返回指定命令的帮助信息
PASV	PASV	请求服务器等待数据连接
QUIT	QUIT	退出系统登录

2. FTP 应答

FTP 应答的标准书写格式为：

```
应答码 <描述信息>
```

FTP 应答由两个部分组成：应答码与描述信息。其中，应答码是由 3 位数字组成的字符串，它是对该应答信息的数字标识，例如 020 表示用户登录成功；描述信息是对应答码的文字描述，例如 020 后面的描述信息是 User login success。表 14-2 给出了常用的 FTP 应答。FTP 客户机与服务器需要按照上述格式实现 FTP 应答，但是描述信息的内容可以由自己来确定。

表 14-2　常用的 FTP 应答

FTP 应答码	格　　式
120	120 Service ready in nnn minutes
125	125 Data connection already open; transfer starting
150	150 File status okay; about to open data connection
220	220 Service ready for new user
225	225 Data connection open; no transfer in progress
226	226 Closing data connection
227	227 Entering Passive Mode (h1,h2,h3,h4,p1,p2)
230	230 User logged in, proceed
250	250 Requested file action okay, completed
331	331 User name okay, need password
332	332 Need account for login
421	421 Service not available, closing control connection
425	425 Can't open data connection
426	426 Connection closed; transfer aborted
450	450 Requested file action not taken
500	500 Syntax error, command unrecognized
501	501 Syntax error in parameters or arguments
530	530 Not logged in
550	550 Requested action not taken

FTP应答的核心部分是开始位置的应答码,它在FTP客户机与服务器编程时不能改变。FTP应答码是根据一定的规则来定义的,通过它可以方便地理解应答信息的类型。应答码的第一个数字表示命令完成状况。其中,1xx表示成功完成命令;2xx表示需要其他信息;3xx表示需要保持数据连接;4xx表示暂时无法完成命令;5xx表示永远无法完成命令。应答码的后两个数字是对应答信息的进一步细分。

图14-5给出了FTP命令与应答的关系。除了LIST命令之外,FTP客户机每发送一个命令,FTP服务器都会返回一个应答。每个FTP命令对应不同的操作结果,都会收到对应的FTP应答。例如,USER命令的应答有230、331、421、500、501与530;PASS命令的应答有230、332、421、500、501与530;PASV命令的应答有227、421、500、501与530;LIST命令的应答有125、150、226、250、421、425、426、450、500、501与530;RETR命令的应答比LIST命令多了550。另外,建立连接相关的应答有120、220与421。

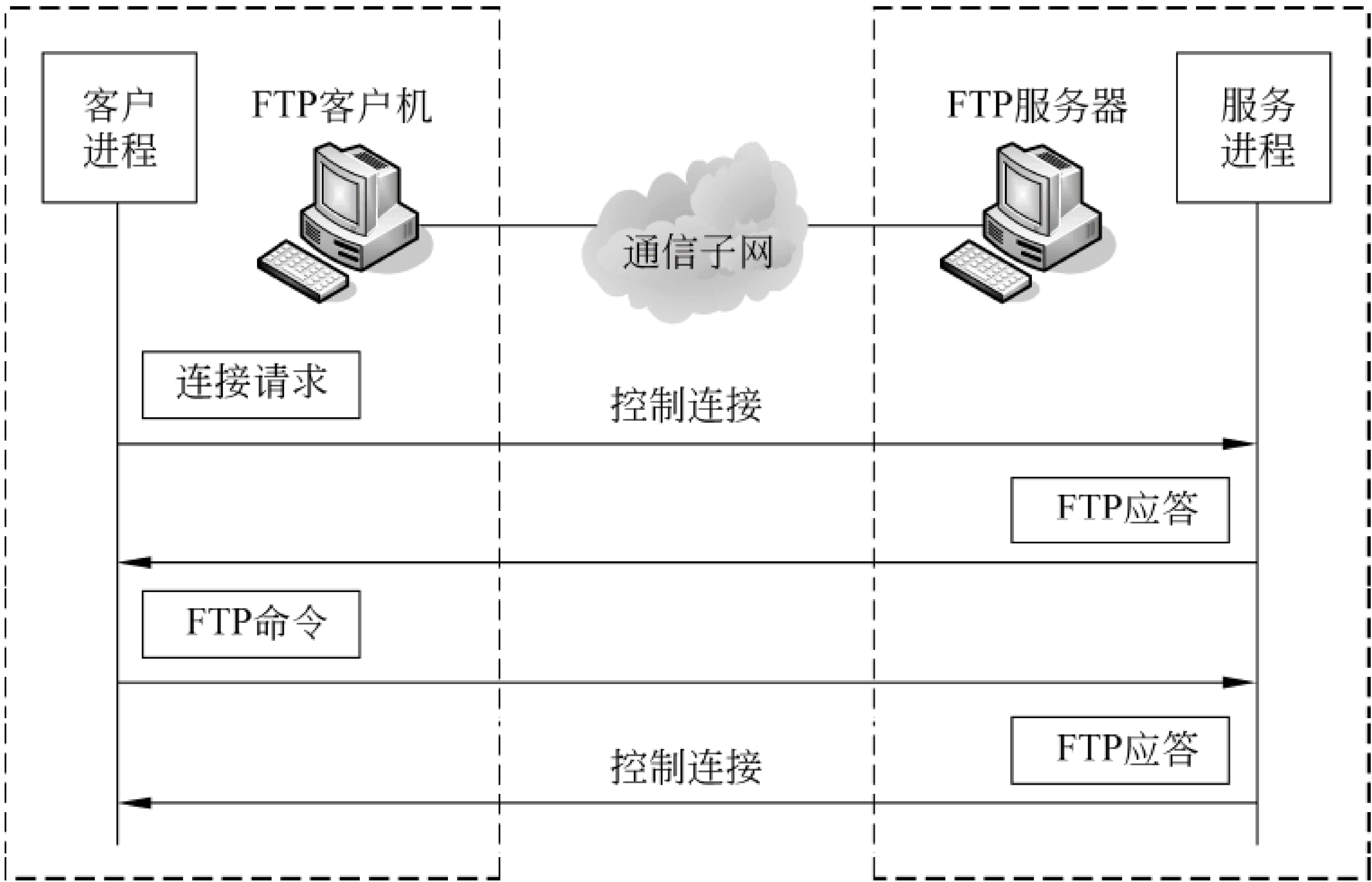


图14-5　FTP命令与应答的关系

14.3　例题分析

14.3.1　设计要求

根据客户机/服务器工作模式,编写FTP客户机程序向服务器发送命令,并将FTP服务器返回的应答信息与数据显示在控制台上。在本练习中为了简便起见,只需要实现USER、PASS、LIST与QUIT命令。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为FtpClient.exe,则程序的命令行格式为:

```
FtpClient server_addr
```

其中,server_addr为FTP服务器的IP地址。

(2) 要求将FTP服务器的状态显示在控制台上,具体格式为:

```
FTP> Control Connection…
响应信息…
FTP> USER:xxxxxx
响应信息…
FTP> PASS:xxxxxx
响应信息…
FTP> LIST
响应信息…
FTP> QUIT
响应信息…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

14.3.2 关键问题

1. 建立控制连接

在FTP客户机与服务器之间进行通信,首先需要建立的就是控制连接。FTP客户机首先调用socket()函数来建立套接字,然后调用connect()函数请求与FTP服务器建立连接,在连接建立后就可以调用send()与recv()函数来发送与接收数据。当FTP客户机向服务器发送连接建立请求后,需要接收与分析FTP服务器返回的应答。FTP服务器返回的应答信息可能包含120、220与421,但是只有接收到的应答码为220时,才表示控制连接成功建立并且可以开始发送FTP命令。

下面给出建立控制连接的伪代码:

```
//套接字异步启动
WSAStartup(MAKEWORD(2,2),&WSAData)
//创建流式 Socket
SocketControl=socket(AF_INET,SOCK_STREAM,0);
//填充本地 Socket 地址
sockaddr_in serveraddr;
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(21);
serveraddr.sin_addr.S_un.S_addr=inet_addr(argv[1]);
//向 FTP 服务器发送 Connect 请求
connect(SocketControl,(sockaddr*)&serveraddr,sizeof(serveraddr));
//从 FTP 服务器获得 Connect 应答
if(RecvReply())
{
    //判断 Connect 请求的应答码
    if(nReplyCode==220)
```

```
        …
    else
        closesocket(SocketControl);
}
```

2. 登录到FTP服务器

FTP客户机与服务器之间建立控制连接后,还需要通过身份认证登录到FTP服务器,这时要验证用户名与密码的正确性。登录FTP服务器需要使用USER与PASS命令,它们分别用来输入FTP用户名与密码。USER与PASS命令是按照规定顺序出现的。FTP客户机向服务器发送USER命令,FTP服务器返回的应答可能包含230、331、421、500、501与530,但是只有接收到的应答码为331时,才表示用户名正确并继续输入密码;FTP客户机向服务器发送PASS命令,FTP服务器返回的应答信息可能包含230、332、421、500、501与530,但是只有接收到的应答码为230时,才表示用户名与密码均正确且成功登录。

下面给出登录到FTP服务器的伪代码:

```
//构造标准格式的USER命令
memcpy(Command,"USER",strlen("USER"));
memcpy(Command+strlen("USER"),CmdBuf,strlen(CmdBuf));
memcpy(Command+strlen("USER")+strlen(CmdBuf),"\r\n",2);
//向FTP服务器发送USER命令
SendCommand();
//从FTP服务器接收应答信息
if(RecvReply())
{
    //判断USER命令的应答码
    if(nReplyCode==230||nReplyCode==331)
        …
}
if(nReplyCode==331)
{
    //构造标准格式的PASS命令
    …
    //向FTP服务器发送PASS命令
    SendCommand()
    //从FTP服务器接收应答信息
    if(RecvReply())
        //判断PASS命令的应答码
        if(nReplyCode==230)
            …
}
```

3. 执行LIST命令

LIST命令用来返回当前目录中的信息(包括子目录与文件),它需要使用数据连接来传输目录信息,因此FTP客户机要与服务器建立数据连接。这时,有两种建立数据连接的

方法：一种是使用PORT命令；另一种是使用PASV命令。其中，PORT命令方式称为主动模式，FTP客户机指定自己用于数据连接的端口，由FTP服务器来与客户机建立数据连接；PASV命令方式称为被动模式，FTP服务器在应答信息中指出用于数据连接的端口，由FTP客户机与服务器建立数据连接。由于很多内部网络不支持PORT方式，因此这里采用的是PASV方式。LIST命令执行完以后，需要释放数据连接。

下面给出执行LIST命令的伪代码：

```
//通过 PASV 命令获得服务器端口
SendCommand();
RecvReply();
//获取 FTP 服务器的数据端口
…
//创建数据连接 Socket
SocketData=socket(AF_INET,SOCK_STREAM,0);
//向 FTP 服务器发送 Connect 请求
connect(SocketData,(sockaddr*)&serveraddr,sizeof(serveraddr));
//向 FTP 服务器发送 LIST 命令
SendCommand();
//从 FTP 服务器接收应答信息
if(RecvReply())
{
    //判断 LIST 命令的应答码
    if(nReplyCode==125||nReplyCode==150||nReplyCode==226)
        …
}
//获得 LIST 命令的目录信息
while(true)
{
    memset(ListBuf,0,MAX_SIZE);
    nRecv=recv(SocketData,ListBuf,MAX_SIZE,0);
}
//关闭数据连接
closesocket(SocketData);
```

4. 程序流程图

图14-6给出了主程序流程图。这里，要求输入的命令行参数必须正确，除了程序本身的名称以外，还需要提供一个FTP服务器的地址。如果命令行参数的个数不是一个，则程序在输出错误信息后退出。如果FTP客户机接收的连接应答码有误，或者接收的USER、PASS、PASV、LIST、QUIT应答码有误，则程序在输出错误信息后退出。

14.3.3 程序源代码

下面给出FTP客户机程序的源代码：

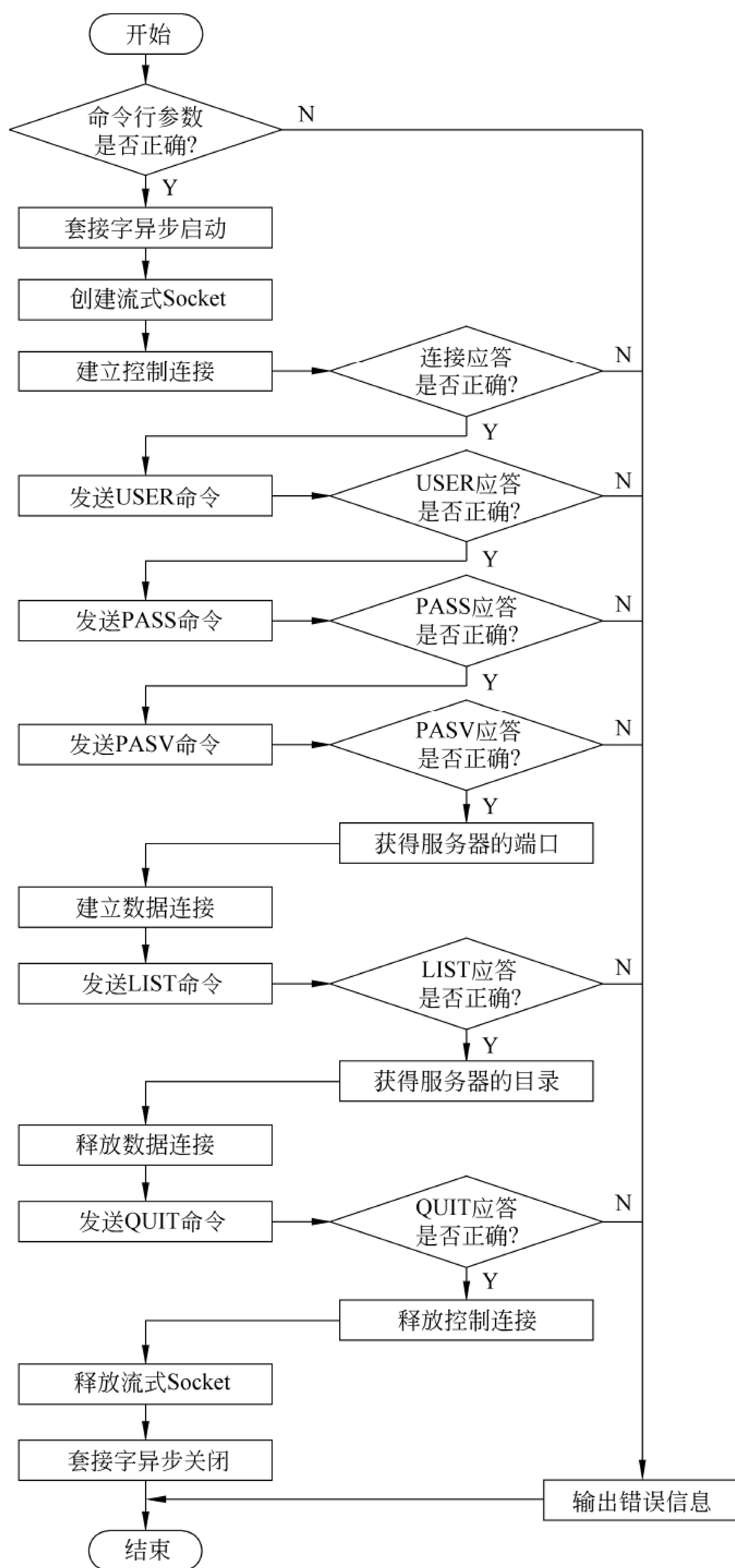

开始
命令行参数是否正确?
N
Y
套接字异步启动
创建流式Socket
建立控制连接
连接应答是否正确?
发送USER命令
USER应答是否正确?
发送PASS命令
PASS应答是否正确?
发送PASV命令
PASV应答是否正确?
获得服务器的端口
建立数据连接
发送LIST命令
LIST应答是否正确?
获得服务器的目录
释放数据连接
发送QUIT命令
QUIT应答是否正确?
释放控制连接
释放流式Socket
套接字异步关闭
输出错误信息
结束

图 14-6　主程序流程图

```
//FtpClient.cpp：定义控制台应用程序的入口点

#include "stdafx.h"
#include "conio.h"
#include "string.h"
#include "winsock2.h"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib
#define MAX_SIZE 4096

char CmdBuf[MAX_SIZE];                           //输入缓冲区
char Command[MAX_SIZE];                          //客户机命令
char ReplyMsg[MAX_SIZE];                         //服务器应答
int nReplyCode;                                  //服务器应答码
bool bConnected=false;                           //是否已登录
SOCKET SocketControl;                            //控制连接套接字
SOCKET SocketData;                               //数据连接套接字

bool RecvReply()                                 //从 FTP 服务器接收应答
{
    int nRecv;
    memset(ReplyMsg,0,MAX_SIZE);
    nRecv=recv(SocketControl,ReplyMsg,MAX_SIZE,0);
    if(nRecv==SOCKET_ERROR)                      //控制连接接收数据
    {
        cout<<endl<<"Socket Recv 失败!"<<endl;
        closesocket(SocketControl);
        return false;
    }
    if(nRecv>4)                                  //获得服务器应答
    {
        char* ReplyCodes=new char[3];
        memset(ReplyCodes,0,3);
        memcpy(ReplyCodes,ReplyMsg,3);
        nReplyCode=atoi(ReplyCodes);
    }
    return true;
}

bool SendCommand()                               //向 FTP 服务器发送命令
{
    int nSend;
```

```
        nSend=send(SocketControl,Command,strlen(Command),0);
        if(nSend==SOCKET_ERROR)                      //控制连接发送数据
        {
            cout<<endl<<"Socket Send 失败!"<<endl;
            return false;
        }
        return true;
    }

    bool DataConnect(char* ServerAddr)
    {
        //向 FTP 服务器发送 PASV 命令
        memset(Command,0,MAX_SIZE);
        memcpy(Command,"PASV",strlen("PASV"));
        memcpy(Command+strlen("PASV"),"\r\n",2);
        if(!SendCommand())                            //发送 PASV 命令
            return false;

        //获得 PASV 命令的应答信息
        if(RecvReply())
        {
            if(nReplyCode!=227)                       //判断 PASV 应答码
            {
                cout<<endl<<"PASV 命令应答出错!"<<endl;
                closesocket(SocketControl);
                return false;
            }
        }

        //分离 PASV 命令的应答信息
        char* part[6];
        if(strtok(ReplyMsg,"("))                     //分离"("后字符串
        {
            for(int i=0;i<5;i++)
            {
                part[i]=strtok(NULL,",");             //分离","前后字符串
                if(!part[i])
                    return false;
            }
            part[5]=strtok(NULL,")");                 //分离")"前字符串
            if(!part[5])
                return false;
        }
        else
            return false;
```

```
    //获取 FTP 服务器的数据端口
    unsigned short ServerPort;
    ServerPort=unsigned short((atoi(part[4])<<8)+atoi(part[5]));

    SocketData=socket(AF_INET,SOCK_STREAM,0);
    if(SocketData==INVALID_SOCKET)                    //创建数据连接套接字
    {
        cout<<endl<<"创建 Socket 失败!"<<endl;
        return false;
    }

    sockaddr_in serveraddr;                           //初始化套接字信息
    memset(&serveraddr,0,sizeof(serveraddr));
    serveraddr.sin_family=AF_INET;
    serveraddr.sin_port=htons(ServerPort);
    serveraddr.sin_addr.S_un.S_addr=inet_addr(ServerAddr);

    //向 FTP 服务器发送 Connect 请求
    int nConnect;
    nConnect=connect(SocketData,(sockaddr* )&serveraddr,sizeof(serveraddr));
    if(nConnect==SOCKET_ERROR)                        //与 FTP 服务器建立连接
    {
        cout<<endl<<"Socket Connect 失败!"<<endl;
        return false;
    }
    return true;
}

void main(int argc,char* argv[])
{
    if(argc!=2)                                       //检查命令行参数
    {
        cout<<endl<<"请按以下格式输入命令行:FtpClient server_addr"<<endl;
        return;
    }

    if(bConnected==true)                              //是否已连接 FTP 服务器
    {
        cout<<endl<<"已连接 FTP 服务器!"<<endl;
        closesocket(SocketControl);
        return;
    }

    WSADATA WSAData;                                  //套接字异步启动
```

```
if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)
{
    cout<<endl<<"WSAStartup 初始化失败!"<<endl;
    return;
}

SocketControl=socket(AF_INET,SOCK_STREAM,0);
if(SocketControl==INVALID_SOCKET)               //创建控制连接套接字
{
    cout<<endl<<"创建 Socket 失败!"<<endl;
    return;
}

sockaddr_in serveraddr;                         //初始化套接字信息
memset(&serveraddr,0,sizeof(serveraddr));
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(21);
serveraddr.sin_addr.S_un.S_addr=inet_addr(argv[1]);

//向 FTP 服务器发送 Connect 请求
cout<<endl<<"FTP>Control Connect…";
int nConnect;
nConnect=connect(SocketControl,(sockaddr*)&serveraddr,sizeof(serveraddr));
if(nConnect==SOCKET_ERROR)                      //与 FTP 服务器建立连接
{
    cout<<endl<<"Socket Connect 失败!"<<endl;
    return;
}

//获得 Connect 应答信息
if(RecvReply())
{
    if(nReplyCode==220)                         //判断 Connect 应答码
        cout<<endl<<ReplyMsg;
    else
    {
        cout<<endl<<"Connect 应答出错!"<<endl;
        closesocket(SocketControl);             //关闭控制连接
        return;
    }
}

//向 FTP 服务器发送 USER 命令
cout<<"FTP>USER:";
```

```
memset(CmdBuf,0,MAX_SIZE);
cin.getline(CmdBuf,MAX_SIZE,'\n');                //输入用户名

memset(Command,0,MAX_SIZE);
memcpy(Command,"USER",strlen("USER"));
memcpy(Command+strlen("USER"),CmdBuf,strlen(CmdBuf));
memcpy(Command+strlen("USER")+strlen(CmdBuf),"\r\n",2);
if(!SendCommand())                                //发送 USER 命令
    return;

//获得 USER 命令的应答信息
if(RecvReply())
{
    if(nReplyCode==230||nReplyCode==331)
        cout<<ReplyMsg;                           //判断 USER 应答码
    else
    {
        cout<<endl<<"USER 命令应答出错!"<<endl;
        closesocket(SocketControl);
        return;
    }
}

if(nReplyCode==331)
{
    //向 FTP 服务器发送 PASS 命令
    cout<<"FTP>PASS:";
    memset(CmdBuf,0,MAX_SIZE);
    cout.flush();
    for(int i=0;i<MAX_SIZE;i++)
    {
        CmdBuf[i]=getch();                        //输入用户密码
        if(CmdBuf[i]=='\r')
        {
            CmdBuf[i]='\0';
            break;
        }
        else
            cout<<'*';
    }
    cout<<endl;

    memset(Command,0,MAX_SIZE);
    memcpy(Command,"PASS",strlen("PASS"));
```

```
        memcpy(Command+strlen("PASS"),CmdBuf,strlen(CmdBuf));
        memcpy(Command+strlen("PASS")+strlen(CmdBuf),"\r\n",2);
        if(!SendCommand())                          //发送 PASS 命令
            return;

        //获得 PASS 命令的应答信息
        if(RecvReply())
        {
            if(nReplyCode==230)                     //判断 PASS 应答码
                cout<<ReplyMsg;
            else
            {
                cout<<endl<<"PASS 命令应答出错!"<<endl;
                closesocket(SocketControl);
                return;
            }
        }
    }

    //向 FTP 服务器发送 LIST 命令
    cout<<"FTP>LIST"<<endl;
    char FtpServer[MAX_SIZE];
    memset(FtpServer,0,MAX_SIZE);                   //获得 FTP 服务器地址
    memcpy(FtpServer,argv[1],strlen(argv[1]));

    if(!DataConnect(FtpServer))                     //建立数据连接
        return;

    memset(Command,0,MAX_SIZE);
    memcpy(Command,"LIST",strlen("LIST"));
    memcpy(Command+strlen("LIST"),"\r\n",2);
    if(!SendCommand())                              //发送 LIST 命令
        return;

    //获得 LIST 命令的应答信息
    if(RecvReply())
    {
        if(nReplyCode==125||nReplyCode==150||nReplyCode==226)
            cout<<ReplyMsg;                         //判断 LIST 应答码
        else
        {
            cout<<endl<<"LIST 命令应答出错!"<<endl;
            closesocket(SocketControl);
            return;
        }
```

```
    }

    //获得 LIST 命令的目录信息
    int nRecv;
    char ListBuf[MAX_SIZE];                         //目录列表缓冲区
    while(true)
    {
        memset(ListBuf,0,MAX_SIZE);
        nRecv=recv(SocketData,ListBuf,MAX_SIZE,0);
        if(nRecv==SOCKET_ERROR)                     //读取目录信息
        {
            cout<<endl<<"Socket Recv 失败!"<<endl;
            closesocket(SocketData);
            return;
        }
        if(nRecv<=0)
            break;
        cout<<ListBuf;                              //显示目录信息
    }
    closesocket(SocketData);                        //关闭数据连接

    //获得 LIST 命令的应答信息
    if(RecvReply())
    {
        if(nReplyCode==226)
            cout<<ReplyMsg;                         //判断 LIST 应答码
        else
        {
            cout<<endl<<"LIST 命令应答出错!"<<endl;
            closesocket(SocketControl);
            return;
        }
    }

    //向 FTP 服务器发送 QUIT 命令
    cout<<"FTP>QUIT"<<endl;
    memset(Command,0,MAX_SIZE);
    memcpy(Command,"QUIT",strlen("QUIT"));
    memcpy(Command+strlen("QUIT"),"\r\n",2);
    if(!SendCommand())                              //发送 QUIT 命令
        return;

    //获得 QUIT 命令的应答信息
    if(RecvReply())
    {
```

```
            if(nReplyCode==221)                        //判断 QUIT 应答码
            {
                cout<<ReplyMsg;
                bConnected=false;
                closesocket(SocketControl);
                return;
            }
            else
            {
                cout<<endl<<"QUIT 命令应答出错!"<<endl;
                closesocket(SocketControl);
                return;
            }
        }
        WSACleanup();                                  //套接字异步关闭
    }
```

图 14-7 给出了 FTP 客户机的执行过程。程序命令行输入为 FtpClient 10.134.37.128。FTP 客户机向指定服务器(10.134.37.128)的熟知端口发送连接请求,在与服务器建立相应的控制连接与数据连接之后,依次发送 USER、PASS、LIST 与 QUIT 命令,并且将接收到的应答信息与数据显示在控制台上。

```
命令提示符
E:\Test\FtpClient\Debug>FtpClient

请按以下格式输入命令行: FtpClient server_addr

E:\Test\FtpClient\Debug>FtpClient 10.134.37.128

FTP>Control Connect...
220-Wellcome to Home Ftp Server!
220 FTP server ready.
FTP>USER:netlab
331 Password required for netlab.
FTP>PASS:******
230 User netlab logged in.
FTP>LIST
150 Opening data connection for directory list.
drw-rw-rw-   1 ftp      ftp            0 Jun 21  2016 .
drw-rw-rw-   1 ftp      ftp            0 Jun 21  2016 ..
-rwxrwxrwx   1 ftp      ftp     45780896 Jun 21  2016 51.0.2704.103_chrome_insta
ller.exe
-rwxrwxrwx   1 ftp      ftp     39151224 Jun 21  2016 sogou_pinyin_8.0.0.8023_69
89.exe
226 File sent ok
FTP>QUIT
221 Goodbye.
```

图 14-7　FTP 客户机的执行过程

14.4　练　习　题

根据客户机/服务器工作模式,编写 FTP 客户机程序向服务器发送命令,并且将 FTP 服务器返回的应答信息与数据显示在控制台上。在本练习中为了简便起见,只需要实现

USER、PASS、LIST、STOR与QUIT命令。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为FtpClient.exe,则程序的命令行格式为:

```
FtpClient server_addr
```

其中,server_addr为FTP服务器的IP地址。

(2) 要求将FTP服务器的状态显示在控制台上,具体格式为:

```
FTP> Control Connection…
响应信息…
FTP> USER:xxxxxx
响应信息…
FTP> PASS:xxxxxx
响应信息…
FTP> LIST
响应信息…
FTP> STOR
响应信息…
FTP> QUIT
响应信息…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第15章 POP客户机程序设计

15.1 设计目的

Internet 提供了很多类型的网络服务,这些服务实际上都是应用层的服务。网络服务是以客户机/服务器模式工作的。电子邮件是基于 TCP 协议的网络服务。本章练习的目的是,通过 POP 客户机程序的设计,了解电子邮件的基本概念与主要功能,掌握应用层服务的设计思路与编程方法。

15.2 相关知识

本章涉及的相关知识包括应用层的概念、电子邮件的概念与工作原理,以及 POP 命令与应答格式。

15.2.1 电子邮件的基本概念

电子邮件服务又称为 E-mail 服务,是指用户通过 Internet 收发电子形式的邮件。电子邮件是一种非常方便、快速和廉价的通信手段,这些都是电子邮件所具有的基本特点。在传统通信中需要几天才能完成的投递,电子邮件服务仅用几分钟、甚至几秒钟就能完成。目前,电子邮件已经成为网络用户的常用通信手段之一,全世界每时每刻都有数以亿计人次通过电子邮件进行通信。早期的电子邮件只能够传输文本信息,当前的电子邮件还可以传输 HTML 格式信息。

电子邮件是伴随着 Internet 发展起来的。1971 年,电子邮件诞生于美国马萨诸塞州的 BBN 公司,该公司受聘于美国军方参与 ARPANET 的建设与维护。电子邮件的发明者是 BBN 公司的 Ray Tomlinson,他在对已有的文件传输程序的基础上,开发出在 ARPANET 中收发信息的电子邮件程序。为了让人们拥有易于识别的电子邮件地址,汤姆林森决定用@符号隔开用户名与邮件服务器地址,这就是现在使用的电子邮件地址的起源。

由于最初的 ARPANET 节点数很少,当时没有多少人使用电子邮件,这种情况直到 ARPANET 转向 Internet 才得到改变。最初,电子邮件受到网络传输速度的限制,那时用户只能发送一些简短的信息,根本无法像现在这样发送多媒体信息。1988 年,第一个图形界面的邮件客户机软件问世,这就是著名的 Euroda 软件。后来,Netscape 与 Microsoft 公

司相继推出邮件客户机软件。随着 Internet 用户数量的急剧增加，电子邮件逐渐成为一种流行的 Internet 服务。

电子邮件系统与现实中的邮政系统具有相似的结构。图 15-1 给出了两种邮件系统的区别。两者之间的不同点主要在于：邮政系统是由人工控制各种运输设备来运转的电子邮件是在 Internet 中通过计算机、应用软件与协议来运转的。电子邮件系统中同样需要有邮局与邮箱，它们就是电子邮件服务器（Mail Server）与电子邮箱（Mail Box）。其中，邮件服务器负责发送与接收电子邮件；电子邮箱负责存储电子邮件。另外，还需要规定电子邮件的书写格式与传输协议。

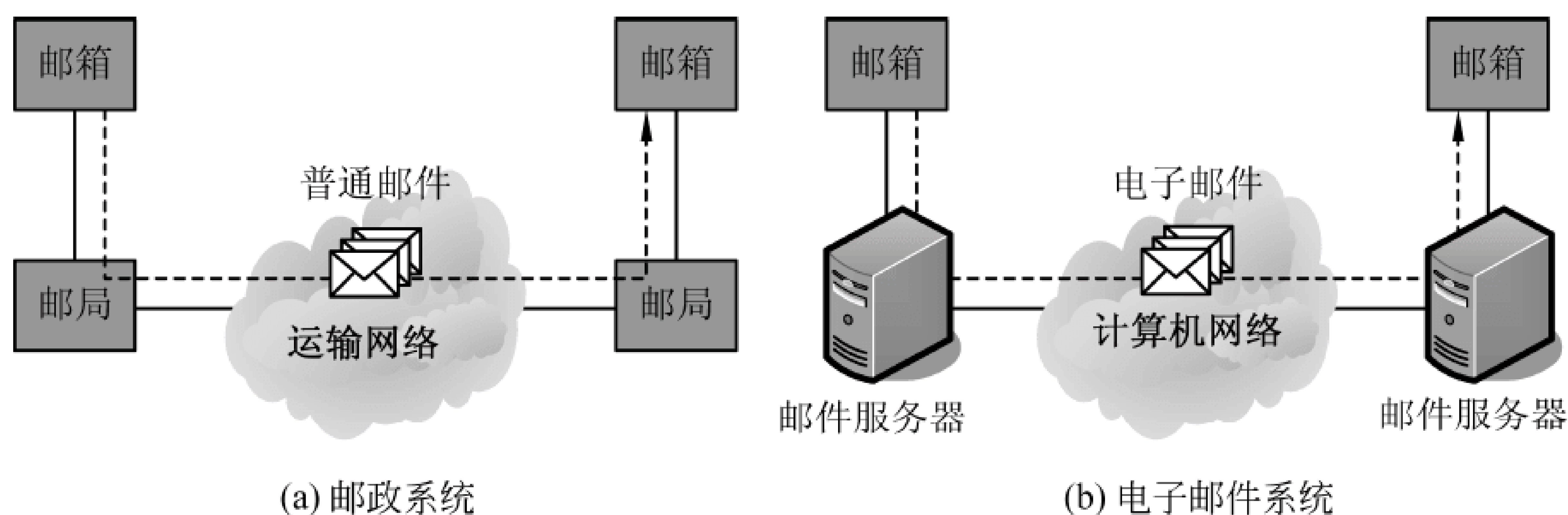


图 15-1　两种邮件系统的区别

邮件服务器是整个电子邮件系统的核心。邮件服务器的主要功能包括：接收发件人通过客户机软件发送的电子邮件，并按照收件人的地址转发到对方的邮件服务器中；接收其他邮件服务器发送的电子邮件，并按照收件人的地址存储在对方的邮箱中；根据收件人的要求将电子邮件发送给收件人的客户机软件。邮箱由提供电子邮件服务的机构来建立，它通常被称为电子邮件账号（包括电子邮件地址与密码）。电子邮件的主要缺点来自于它的安全性相对较弱，通常邮件内容是以明文形式存储与传输。

15.2.2　邮件服务的工作原理

电子邮件服务在传输层采用 TCP 协议，在传输邮件之前需要先建立连接，要经过建立连接、传输数据与释放连接的基本过程。电子邮件服务在应用层采用多种协议，分别用于实现电子邮件的发送与接收。这些应用层协议主要包括简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）、邮局协议（Post Office Protocol，POP）和交互式邮件访问协议（Interactive Mail Access Protocol，IMAP）。其中，SMTP 协议用于将邮件从客户机发送到邮件服务器；POP 协议与 IMAP 协议用于将邮件从邮件服务器接收到客户机。

电子邮件服务采用客户机/服务器工作模式。电子邮件服务包括两个组成部分：邮件客户机与邮件服务器。其中，邮件客户机是电子邮件服务的使用者；邮件服务器是电子邮件服务的提供者。电子邮件的发送与接收采用不同的协议，客户机与邮件服务器需要实现两种协议，即 SMTP 协议与 POP 或 IMAP 协议中的一种，目前通常采用的是 POP 协议。图 15-2 给出了电子邮件的工作原理。客户机主要包括两个部分：SMTP 客户机与 POP 客户机。邮件服务器主要包括三个部分：SMTP 服务器、POP 服务器与电子邮箱。

电子邮件被存储在邮件服务器的相应邮箱中。用户通过客户机与邮件服务器建立连接，并向邮件服务器发出发送或接收请求。如果是用于邮件发送的 SMTP 请求，客户机会

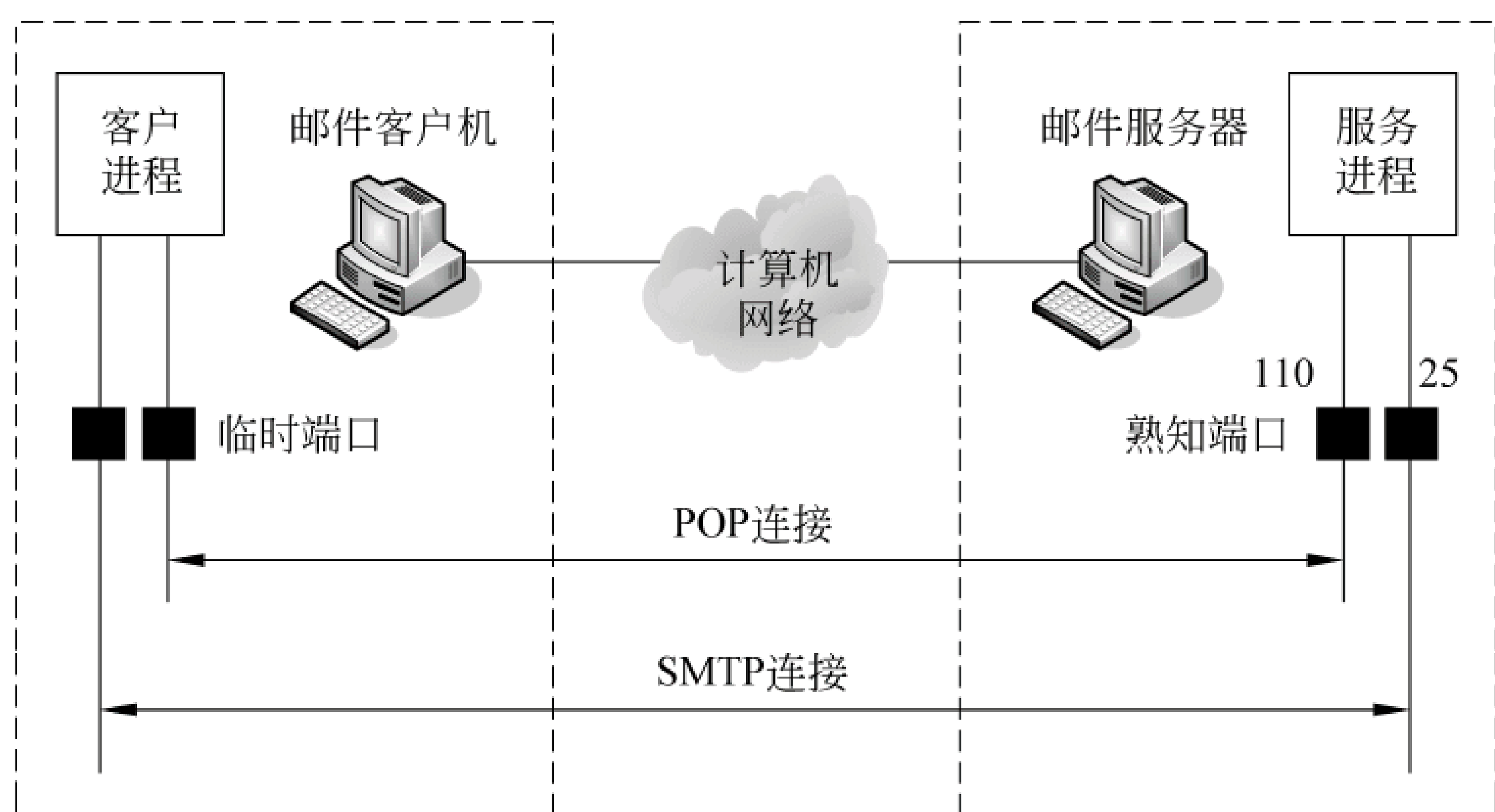


图 15-2　电子邮件的工作原理

将邮件发送到自己的邮件服务器,该邮件服务器将邮件转发到相应的邮箱,整个过程可能经过多个邮件服务器转发;如果是用于邮件接收的 POP 请求,邮件服务器从相应的邮箱中读取邮件,客户机接收返回的邮件后进行解释,并将邮件信息显示在客户机上。

SMTP 协议是电子邮件服务的发送协议,它是 TCP/IP 协议族的应用层协议。1982 年,RFC821 定义了 SMTP 协议的最初版本。2001 年,RFC2821 定义 SMTP 协议的最新版本,它对早期的协议内容没有实质性改动。SMTP 服务器使用的熟知端口号为 25。SMTP 协议详细规定命令与应答的格式。SMTP 请求由客户机发送给邮件服务器,请求类型主要包括 HELO、MAIL、RCPT、DATA 与 QUIT。SMTP 应答由邮件服务器返回给客户机,它可能对应于不同的 SMTP 请求。SMTP 协议规定了 21 个应答状态码。SMTP 请求只能按照特定的顺序发送,通常顺序为 HELO、MAIL、RCPT、DATA 与 QUIT。

POP 协议是电子邮件服务的接收协议之一,它是 TCP/IP 协议族的应用层协议。1984 年,RFC918 文档定义 POP 协议的最初版本。1996 年,RFC1939 文档定义 POP 协议的最新版本,它是 POP 协议的第 3 版(简称 POP3)。目前,大多数电子邮件服务使用的是 POP3 协议。POP3 协议采用持续连接的通信方式。POP3 协议使用的熟知端口号为 110。POP3 协议有两种工作模式:删除模式与保留模式。其中,删除模式在读取邮件后删除邮件;保留模式在读取邮件后仍保留在邮箱中。

IMAP 协议是电子邮件服务的接收协议之一。1994 年,RFC1730 文档定义 IMAP 协议第 4 版(简称 IMAP4)的最初版本。2003 年,RFC3501 文档定义 IMAP4 的最新版本。IMAP4 协议的工作原理与 POP3 协议相似,但是它比 POP3 协议支持更多功能。POP3 协议不支持用户在服务器中整理邮件、定义不同的文件夹,以及在下载邮件之前部分检查邮件内容。IMAP4 协议可以实现上述功能,例如允许用户在下载邮件之前,检查邮件头部与对邮件正文搜索。但是,这就造成 IMAP4 协议结构复杂,难于实现。

15.2.3　邮件地址与邮件格式

电子邮件地址(E-mail Address)是邮件服务器中的邮箱地址。Ray TomLinson 最早提出电子邮件地址,使用@来隔开用户名与邮件服务器地址,后来这种电子邮件地址格式被写入 RFC 文档,使得它成为电子邮件服务方面的标准之一。电子邮件地址的关键是保证每个

地址的唯一性，以便邮件可以经过邮件服务器的转发，被准确地投递到相应的邮箱中。

电子邮件地址的具体格式为：

```
用户名@主机名
```

其中，用户名是用户在邮件服务器中的邮箱名，它在同一个邮件服务器中必须是唯一的；主机名是邮箱所在的邮件服务器的名称，用来标识邮件服务器所在域的位置，这样保证电子邮件地址在全球范围内唯一。

图 15-3 给出了电子邮件地址格式。在这个例子中，电子邮件地址为 island@nankai.edu.cn。其中，island 是邮箱名称；nankai.edu.cn 是邮件服务器名称，它位于为“南开大学”所分配的域中。



图 15-3　电子邮件地址格式

电子邮件与普通的邮政信件相似，也有标准的信件格式方面的规定，以保证邮件能够在不同的邮件服务器之间转发。1982 年，RFC822 文档定义电子邮件的信件格式，它是目前电子邮件仍遵循的信件格式标准。2001 年，RFC2822 文档定义信件格式的最新版本，它并没有对早期的信件格式进行大幅度的改动。SMTP 协议将邮件整个封装在邮件对象中，其中的所有信息都由 ASCII 码组成。按照电子邮件的信件格式规定，电子邮件报文可以由多个报文行组成，各行之间用回车(CR)与换行(LF)符分隔。

图 15-4 给出了电子邮件信件格式。邮件对象包括两个部分：信封与邮件内容。实际上，信封就包含两种 SMTP 请求，用来给出收件人与发件人地址。电子邮件包括两个部分：邮件头(Mail Header)与邮件体(Mail Body)。其中，邮件头由邮件的相关信息构成，其中的部分信息由系统自动生成，例如发信人(From：)、发送时间(Data：)等；其他信息由发件人输入，例如收信人(To：)、邮件主题(Subject：)与抄送人地址(Cc：)等。邮件体是要发送的邮件正文部分。

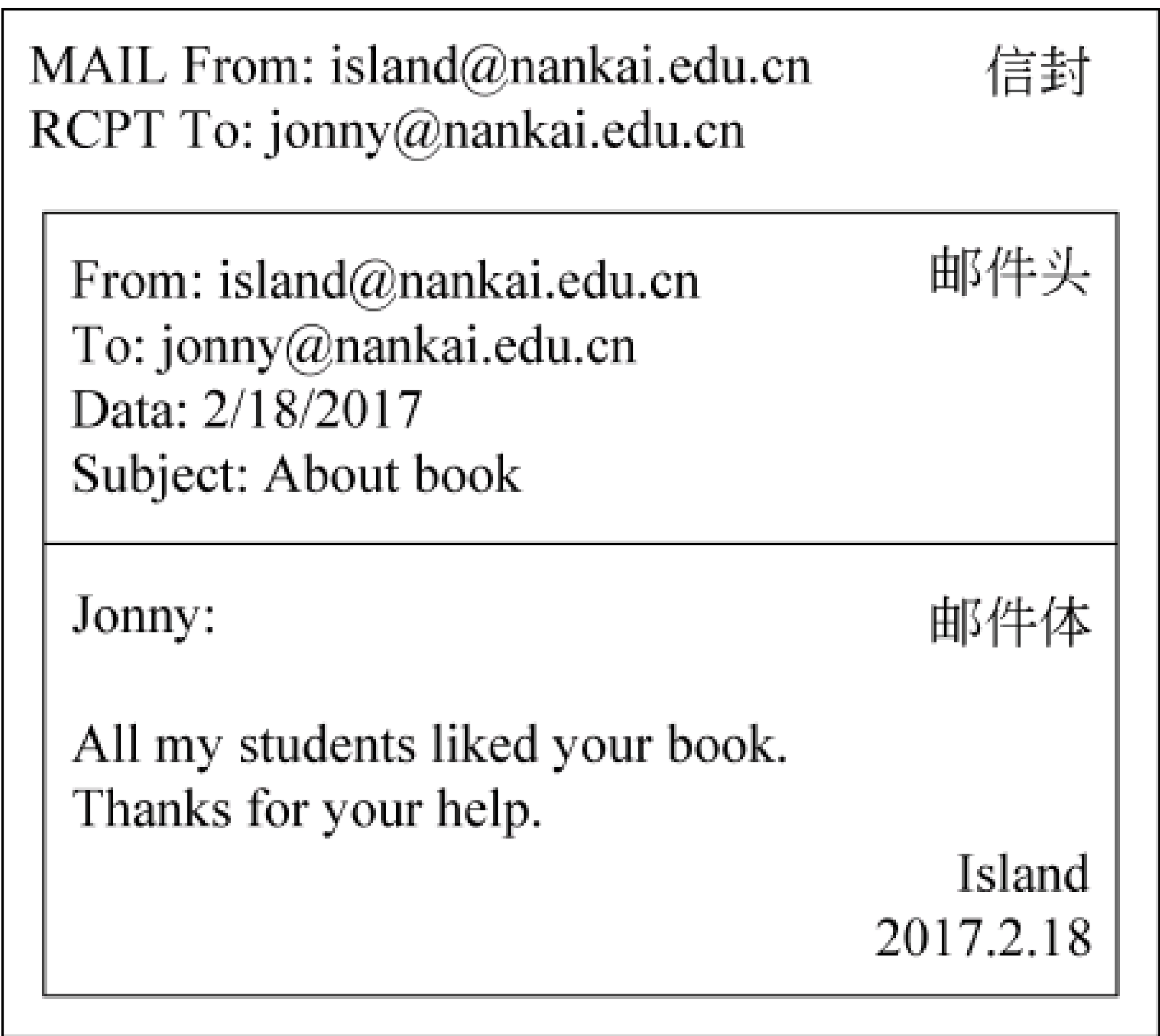


图 15-4　电子邮件信件格式

SMTP 协议只能传输由 ASCII 组成的邮件信息，这样就无法支持那些非 ASCII 码的语种，例如中文、法文、德文与俄文等。另外，它也不支持在邮件中附带二进制文件，例如图片、音频、视频和可执行文件等。1993 年，多用途 Internet 邮件扩展(Multi-purpose Internet Mail Extensions，MIME)出现，它是一种辅助性的邮件编码协议。MIME 可以使非 ASCII

码的数据通过SMTP协议传输,这样使电子邮件的用途变得更加广泛。MIME头部被添加在原始的SMTP头部中,用来定义编码与解码的格式参数。

15.2.4 POP命令与应答

POP客户机与服务器之间传输控制信息,这些信息用于完成具体的POP操作。控制信息可以分为两种类型:POP命令与POP应答。其中,POP命令是POP客户机向服务器发送的操作请求,例如请求从服务器读取邮件内容;POP应答是POP服务器根据操作情况,向客户机返回的应答信息。图15-5给出了POP命令与应答的关系。POP协议详细规定每种协议动作的实现顺序。

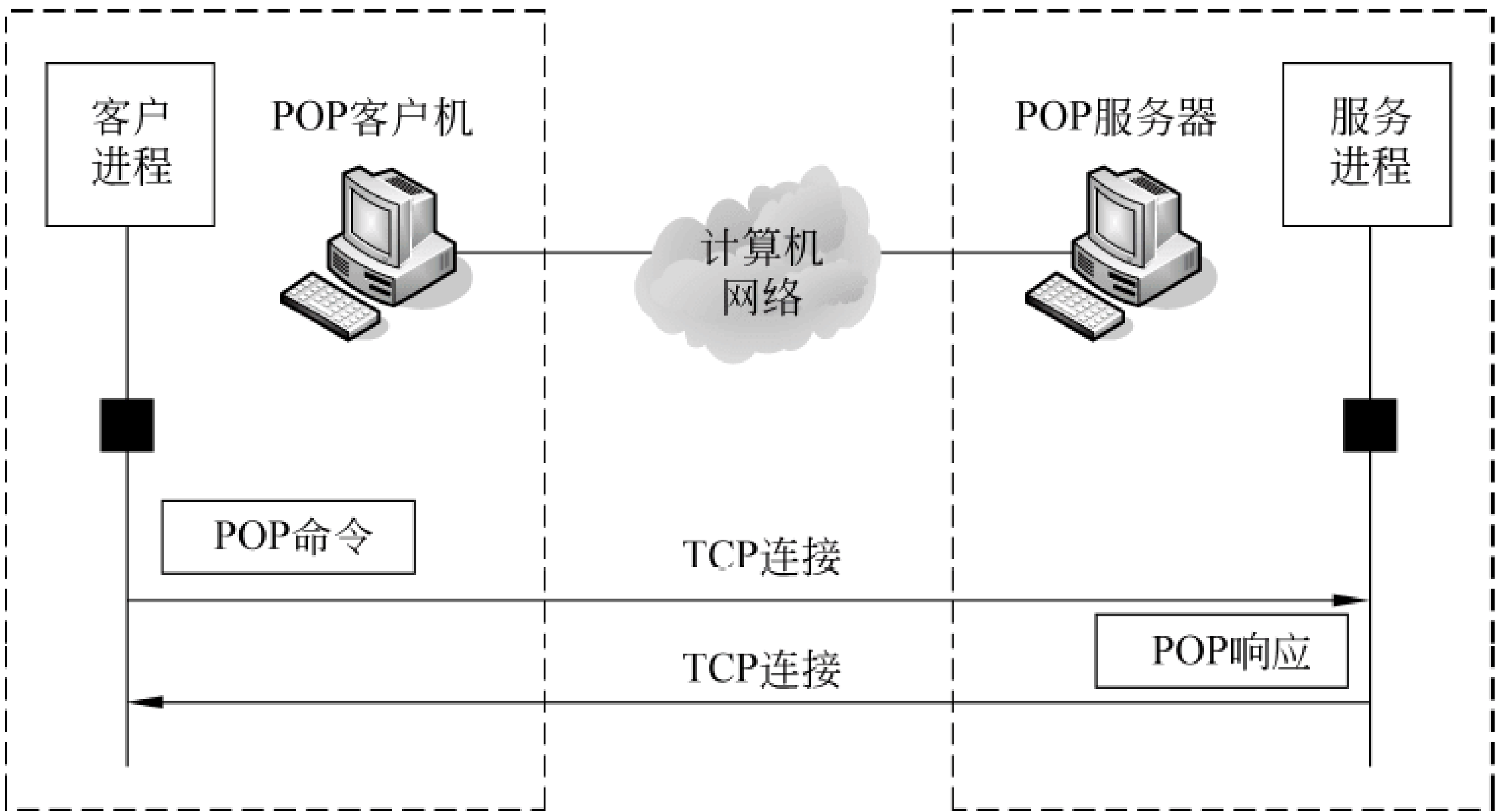


图15-5 POP命令与应答的关系

POP命令的标准格式为:

```
命令名 <参数>
```

POP命令由两个部分组成:命令名与参数。其中,命令名是由3或4个大写字母组成的字符串,它是对该命令的英文描述的缩写,例如USER命令是User Name的缩写;参数是完成命令所需要使用的附加信息,例如USER命令的参数为用户名。所有命令由一对回车(CR)与换行(LF)符来表示结束。表15-1给出了主要的POP命令。其中,最基本的命令包括USER、PASS、STAT、LIST、RETR、DELE、RSET与QUIT。

表15-1 主要的POP命令

命令名	命令格式	命令用途
USER	USER <username>	系统登录需要的用户名
PASS	PASS <password>	系统登录需要的密码
APOP	APOP <name,digest>	系统登录采用经过认证的密码信息
STAT	STAT	返回邮箱统计信息,包括邮件数、总字节数
LIST	LIST <message>	返回指定邮件的大小
RETR	RETR <message>	返回指定邮件的内容,包括邮件头与正文
DELE	DELE <message>	为指定邮件做删除标记,在QUIT时执行
RSET	RSET	撤销所有的DELE命令

续表

命　令　名	命 令 格 式	命 令 用 途
TOP	TOP <message,n>	返回指定邮件的内容的前 n 行
NOOP	NOOP	没有动作,只是等待服务器返回 OK
QUIT	QUIT	退出系统登录

POP 应答的标准格式为:

```
应答码 描述信息
```

POP 应答由两个部分组成:应答码与描述信息。其中,应答码是由+OK 或-ERR 组成的字符串,用来表示 POP 应答的操作状态,例如+OK 表示操作成功,-ERR 表示操作失败;描述信息是对应答码的文字描述,例如 USER 操作成功的描述信息是 valid,USER 操作失败的描述信息是 invalid。所有应答由一对回车(CR)与换行(LF)符来表示结束。例如,POP 应答+OK valid,表示服务器成功执行该操作。

POP 客户机发送每个 POP 命令,POP 服务器返回一个或多个 POP 应答。对应 POP 服务器的不同处理结果,每个 POP 命令有对应的 POP 应答。例如,USER 与 PASS 的应答为+OK valid 与-ERR invalid;STAT 的应答为+OK 2 320 或-ERR;LIST 的应答依次为+OK 2 messages(320 octets)、1 120、2 200 或-ERR no such message;RETR 的应答依次为+OK 120 octets、正文、. 或-ERR no such message;QUIT 的应答为+OK 或-ERR。

15.3 例题分析

15.3.1 设计要求

根据客户机/服务器工作模式,编写 POP 客户机程序向服务器发送命令,并将 POP 服务器返回的应答信息与数据显示在控制台上。在本练习中为了简便起见,只需要实现 USER、PASS、STAT、LIST 与 QUIT 命令。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为 PopClient. exe,则程序的命令行格式为:

```
PopClient server_addr
```

其中,server_addr 为 POP 服务器的域名或 IP 地址。

(2) 要求将 POP 服务器的状态显示在控制台上,具体格式为:

```
POP> Control Connection…
应答信息…
POP> USER:xxxxxx
应答信息…
POP> PASS:xxxxxx
应答信息…
```

```
POP> STAT
应答信息…
POP> LIST
应答信息…
POP> QUIT
应答信息…
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

15.3.2 关键问题

1. 建立数据连接

在 POP 客户机与服务器之间通信,第一步是建立两者之间的 TCP 连接。首先,调用 socket()函数建立套接字,SOCK_STREAM 表示流式套接字;然后,填充服务器的套接字地址,其中的地址填写 POP3 服务器的域名或 IP 地址,端口填写 POP3 的熟知端口 110;接着,调用 connect()函数请求与服务器建立连接;最后,接收与分析服务器返回的应答信息,可能返回的应答信息为+OK 或-ERR,但是只有接收的应答码为+OK 时,表示这个连接已经成功建立,可以通过该连接发送 POP 命令。

下面给出建立数据连接的伪代码:

```
//创建流式套接字
Socket=socket(AF_INET,SOCK_STREAM,0);
//判断 IP 地址或 POP 服务器名
if(argv[1]不是 IP 地址)
    hostent * pHostent=gethostbyname(argv[1]);
//填充服务器的套接字地址
sockaddr_in serveraddr;
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(110);
serveraddr.sin_addr.s_addr=IP 地址;
//向 POP 服务器发送建立连接请求
connect(Socket,(sockaddr *)&serveraddr,sizeof(serveraddr));
//从 POP 服务器获得应答信息
recv(Socket,Respond,MAX_SIZE,0);
//从应答消息中解析 POP 应答码
…
```

2. 登录到 POP 服务器

POP 客户机与服务器建立 TCP 连接后,需要通过身份认证以登录 POP 服务器,这时要验证用户名与密码的正确性。登录 POP 服务器使用 USER 与 PASS 命令,它们分别用来输入用户名与密码。USER 与 PASS 命令是按照规定顺序出现的。POP 客户机向服务

器发送 USER 命令，POP 服务器可能返回的应答为＋OK 或-ERR，当接收的应答码为＋OK 时，表示用户名正确并且需要继续输入密码；POP 客户机向服务器发送 PASS 命令，POP 服务器可能返回的应答为＋OK 或-ERR，当接收到的应答码为＋OK 时，表示用户名与密码均正确并已成功登录。

下面给出登录到 POP 服务器的伪代码：

```
//构造标准的 USER 命令
memcpy(Command,"USER",strlen("USER"));
memcpy(Command+strlen("USER"),m_Account,strlen(m_Account));
//向 POP 服务器发送 USER 命令
send(Socket,Command,strlen(Command),0);
//从 POP 服务器获得应答信息
recv(Socket,Respond,MAX_SIZE,0);
从应答消息中解析 POP 应答码
if(POP 应答码==+OK)
{
    输出 POP 服务器的应答信息
    //构造标准的 PASS 命令
    memcpy(Command,"PASS",strlen("PASS"));
    memcpy(Command+strlen("PASS"),m_Password,strlen(m_Password));
    //向 POP 服务器发送 PASS 命令
    send(Socket,Command,strlen(Command),0);
    //从 POP 服务器获得应答信息
    recv(Socket,Respond,MAX_SIZE,0);
    从应答消息中解析 POP 应答码
    if(POP 应答码==+OK)
        输出 POP 服务器的应答信息
}
```

3. 接收邮件列表

如果需要接收邮箱中的指定邮件，首先需要获得邮箱中的邮件列表。STAT 命令用来返回邮箱的统计信息，包括邮件数量与字节总数。POP 客户机向服务器发送 STAT 命令，POP 服务器可能返回的应答为＋OK 或-ERR，当接收的应答码为＋OK 时，应答信息的后两个值依次为邮件数量与字节总数。LIST 命令用来返回邮箱中的邮件列表。POP 客户机向服务器发送 LIST 命令，POP 服务器可能返回的应答为＋OK 或-ERR，当接收的应答码为＋OK 时，后面接收的是邮件列表，直到接收到句点“.”为止。

下面给出接收邮件列表的伪代码：

```
//构造标准的 STAT 命令
memcpy(Command,"STAT",strlen("STAT"));
//向 POP 服务器发送 STAT 命令
send(Socket,Command,strlen(Command),0);
//从 POP 服务器获得应答信息
```

```
recv(Socket,Respond,MAX_SIZE,0);
从应答消息中解析 POP 应答码
if(POP 应答码==+OK)
    输出 POP 服务器的应答信息
//构造标准的 LIST 命令
memcpy(Command,"LIST number",strlen("LIST number"));
//向 POP 服务器发送 LIST 命令
send(Socket,Command,strlen(Command),0);
//从 POP 服务器获得应答信息
recv(Socket,Respond,MAX_SIZE,0);
从应答消息中解析 POP 应答码
if(POP 应答码==+OK)
{
    输出 POP 服务器的应答信息
    输出接收到的邮件列表
}
```

4. 程序流程图

图 15-6 给出了主程序流程图。这里,要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要提供一个 POP 服务器的地址。如果命令行参数的个数不是一个,则程序在输出错误信息后退出。如果 POP 客户机接收的连接应答码有误,或者接收的 USER、PASS、STAT、LIST、QUIT 应答码有误,则程序在输出错误信息后退出。

15.3.3 程序源代码

下面给出 POP 客户机程序的源代码:

```
//PopClient.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "conio.h"
#include "string.h"
#include "winsock2.h"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                    //加载 ws2_32.lib
#define MAX_SIZE 4096

char CmdBuf[MAX_SIZE];
char Command[MAX_SIZE];
char Respond[MAX_SIZE];
bool RespondStatue;
```

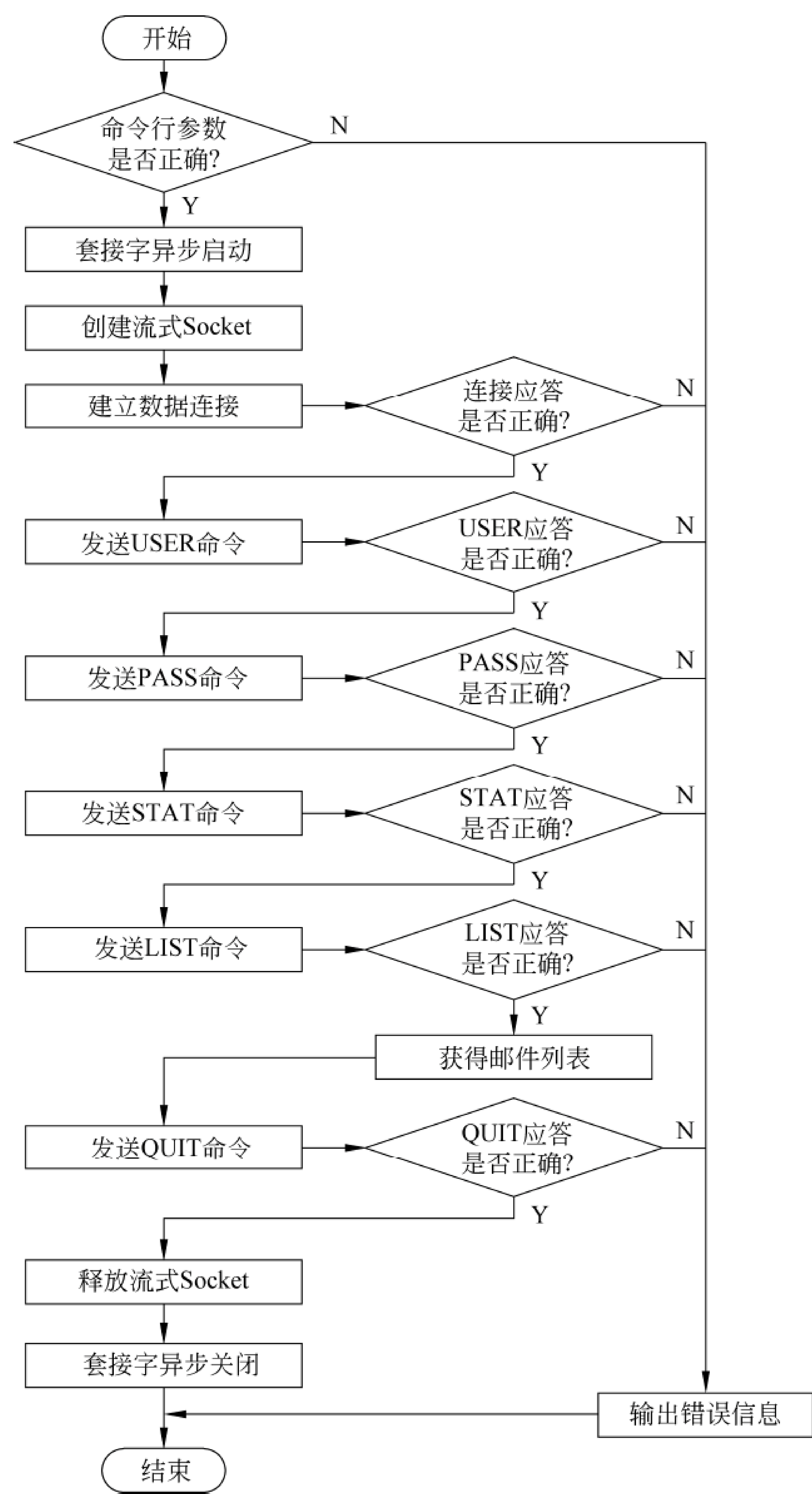


图 15-6　主程序流程图

```
SOCKET sock;

bool SendCommand()                                  //向 POP 服务器发送命令
{
    int nSend;
    nSend=send(sock,Command,strlen(Command),0);
    if(nSend==SOCKET_ERROR)                         //通过套接字发送数据
    {
        cout<<endl<<"Socket Send 失败!"<<endl;
```

```
            return false;
        }
        return true;
    }

    bool RecvReply()                                    //从 POP 服务器接收应答
    {
        int nRecv;
        memset(Respond,0,MAX_SIZE);
        nRecv=recv(sock,Respond,MAX_SIZE,0);
        if(nRecv==SOCKET_ERROR)                         //通过套接字接收数据
        {
            cout<<endl<<"Socket Recv 失败!"<<endl;
            return false;
        }
        if(memcmp(Respond,"+OK",3)==0)                  //从应答中解析应答状态
            RespondStatue=true;
        else
            RespondStatue=false;
        return true;
    }

    void main(int argc,char * argv[])
    {
        if(argc!=2)                                     //检查命令行参数
        {
            cout<<endl<<"请按以下格式输入命令行:PopClient server_addr"<<endl;
            return;
            }

        WSADATA WSAData;                                //套接字异步启动
        if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)
        {
            cout<<endl<<"WSAStartup 初始化失败!";
            return;
        }

        sock=socket(AF_INET,SOCK_STREAM,0);
        if(sock==INVALID_SOCKET)                        //创建流式套接字
        {
            cout<<endl<<"创建 Socket 失败!";
            return;
        }

        int ipaddress;                                  //判断域名或 IP 地址
```

```
ipaddress=inet_addr(argv[1]);
if(ipaddress==INADDR_NONE)
{
    hostent * pHostent=gethostbyname(argv[1]);
    if(pHostent)
        ipaddress=(*(in_addr*)pHostent->h_addr).s_addr;
}

sockaddr_in serveraddr;                       //初始化套接字信息
memset(&serveraddr,0,sizeof(serveraddr));
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(110);
serveraddr.sin_addr.S_un.S_addr=ipaddress;

//向 POP3 服务器发送 Connect 请求
cout<<endl<<"POP3>Connect to "<<argv[1]<<"…";
int nConnect;
nConnect=connect(sock,(sockaddr*)&serveraddr,sizeof(serveraddr));
if(nConnect==SOCKET_ERROR)                    //请求建立数据连接
{
    cout<<endl<<"Socket Connect 失败!";
    return;
}

//获得 Connect 应答信息
if(!RecvReply())
    return;
else
{
    if(RespondStatue==true)                   //判断连接应答状态
        cout<<endl<<Respond;
    else
    {
        cout<<endl<<"应答状态出错!";
        return;
    }
}

//向 POP3 服务器发送 USER 命令
cout<<"POP3>USER:";
memset(CmdBuf,0,MAX_SIZE);
cin.get(CmdBuf,MAX_SIZE);                     //输入用户名

memset(Command,0,MAX_SIZE);
```

```
memcpy(Command,"USER",strlen("USER"));
memcpy(Command+strlen("USER"),CmdBuf,strlen(CmdBuf));
memcpy(Command+strlen("USER")+strlen(CmdBuf),"\r\n",2);
if(!SendCommand())                              //发送 USER 命令
    return;

//获得 USER 命令的应答信息
if(!RecvReply())
    return;
else
{
    if(RespondStatue==true)                     //判断 USER 应答状态
        cout<<Respond;
    else
    {
        cout<<"应答状态出错!";
        return;
    }
}

//向 POP3 服务器发送 PASS 命令
cout<<"POP3>PASS:";
memset(CmdBuf,0,MAX_SIZE);
cout.flush();
for(int i=0;i<MAX_SIZE;i++)
{
    CmdBuf[i]=getch();                          //输入用户密码
    if(CmdBuf[i]=='\r')
    {
        CmdBuf[i]='\0';
        break;
    }
    else
        cout<<'*';
}

memset(Command,0,MAX_SIZE);
memcpy(Command,"PASS",strlen("PASS"));
memcpy(Command+strlen("PASS"),CmdBuf,strlen(CmdBuf));
memcpy(Command+strlen("PASS")+strlen(CmdBuf),"\r\n",2);
if(!SendCommand())                              //发送 PASS 命令
    return;

//获得 PASS 命令的应答信息
```

```
if(!RecvReply())
    return;
else
{
    if(RespondStatue==true)                    //判断 PASS 应答状态
        cout<<endl<<Respond;
    else
    {
        cout<<endl<<"应答状态出错!";
        return;
    }
}

//向 POP3 服务器发送 STAT 命令
cout<<"POP3>STAT";
memset(Command,0,MAX_SIZE);
memcpy(Command,"STAT",strlen("STAT"));
memcpy(Command+strlen("STAT"),"\r\n",2);
if(!SendCommand())                             //发送 STAT 命令
    return;

//获得 STAT 命令的应答信息
if(!RecvReply())
    return;
else
{
    if(RespondStatue==true)                    //判断 STAT 应答状态
        cout<<endl<<Respond;
    else
    {
        cout<<endl<<"应答状态出错!";
        return;
    }
}

//向 POP3 服务器发送 LIST 命令
cout<<"POP3>LIST";
memset(CmdBuf,0,MAX_SIZE);
cin.get(CmdBuf,MAX_SIZE);                      //输入邮件信息

memset(Command,0,MAX_SIZE);
memcpy(Command,"LIST",strlen("LIST"));
memcpy(Command+strlen("LIST"),CmdBuf,strlen(CmdBuf));
memcpy(Command+strlen("LIST")+strlen(CmdBuf),"\r\n",2);
```

```
    if(!SendCommand())                          //发送 LIST 命令
        return;

    //获得 LIST 命令的应答信息
    if(!RecvReply())
        return;
    else
    {
        if(RespondStatue==true)                 //判断 LIST 应答状态
            cout<<endl<<Respond;
        else
        {
            cout<<endl<<"应答状态出错!";
            return;
        }
    }

    //向 POP3 服务器发送 QUIT 命令
    cout<<"POP3>QUIT";
    memset(Command,0,MAX_SIZE);
    memcpy(Command,"QUIT",strlen("QUIT"));
    memcpy(Command+strlen("QUIT"),"\r\n",2);
    if(!SendCommand())                          //发送 QUIT 命令
        return;

    //获得 QUIT 命令的应答信息
    if(!RecvReply())
        return;
    else
    {
        if(RespondStatue==true)                 //判断 QUIT 应答状态
            cout<<endl<<Respond;
        else
        {
            cout<<endl<<"应答状态出错!";
            return;
        }
    }

    closesocket(sock);                          //关闭流式 Socket
    WSACleanup();                               //套接字异步关闭
}
```

图15-7给出了POP客户机的执行过程。程序命令行输入为PopClient pop.163.com。POP客户机向指定服务器(pop.163.com)的熟知端口发送连接请求,在与服务器建立相应

的连接之后，依次发送 USER、PASS、STAT、LIST 与 QUIT 命令，并且将接收到的应答信息与数据显示在控制台上。

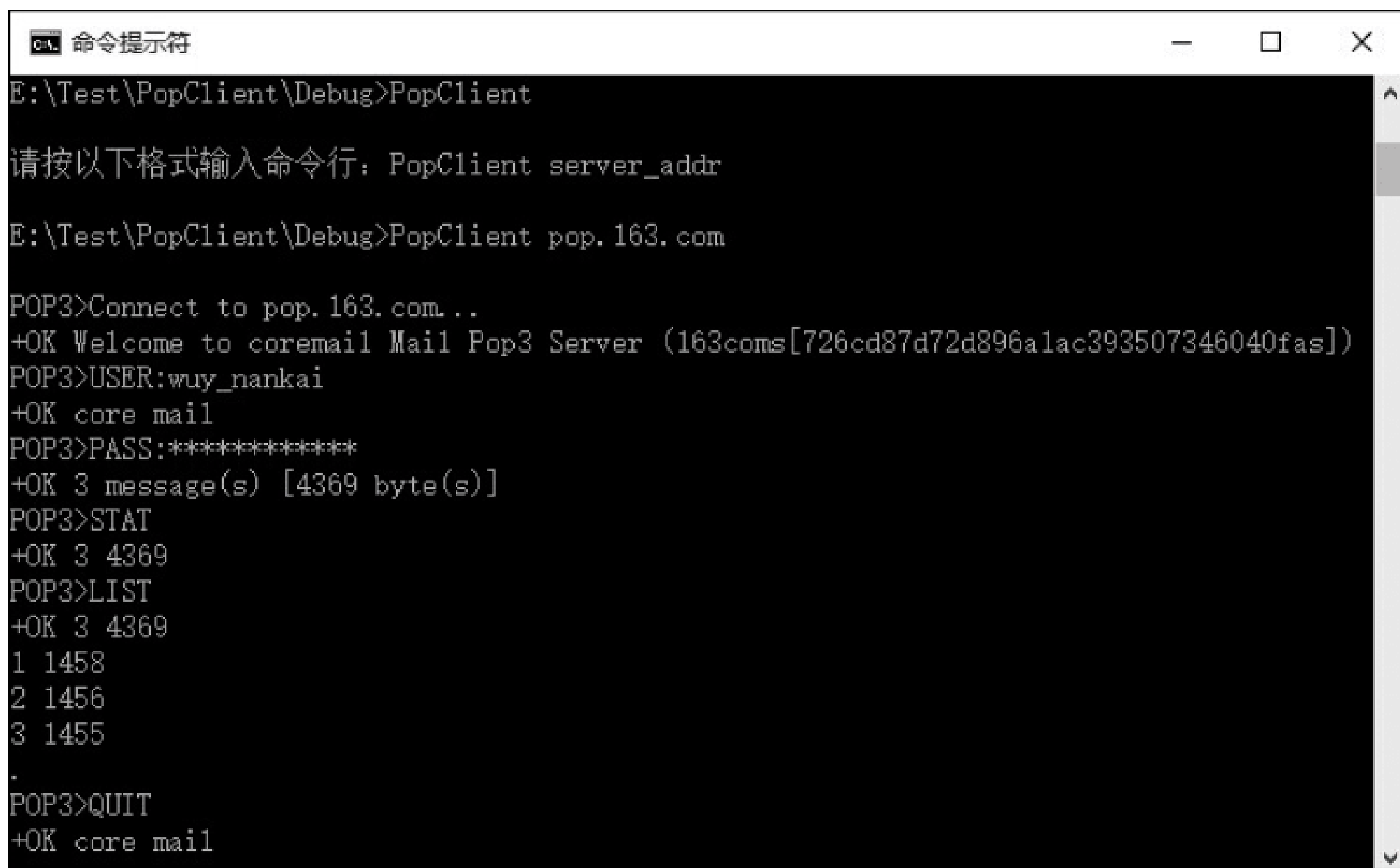


图 15-7　POP 客户机的执行过程

15.4 练　习　题

根据客户机/服务器工作模式，编写 POP 客户机程序向服务器发送命令，并且将 POP 服务器返回的应答信息与数据显示在控制台上。在本练习中为了简便起见，只需要实现 USER、PASS、STAT、RETR 与 QUIT 命令。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如，可执行文件名为 PopClient.exe，则程序的命令行格式为：

```
PopClient server_addr
```

其中，server_addr 为 POP 服务器的域名或 IP 地址。

(2) 要求将 POP 服务器的状态显示在控制台上，具体格式为：

```
POP> Control Connection…
应答信息…
POP> USER:xxxxxx
应答信息…
POP> PASS:xxxxxx
应答信息…
POP> STAT
应答信息…
POP> RETR
应答信息…
邮件内容…
```

```
POP> QUIT
应答信息…
```


(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

第 16 章 包过滤防火墙程序设计

16.1 设 计 目 的

随着计算机网络的普及与广泛应用,网络安全问题越来越得到重视。防火墙是网络安全技术中的重要组成部分。本章练习的目的是,通过包过滤防火墙程序设计,了解防火墙的基本概念与主要功能,掌握网络层包过滤技术的设计思路与编程方法。

16.2 相 关 知 识

本章涉及的相关知识包括网络安全的重要性、防火墙的概念与包过滤工作原理。

16.2.1 网络安全的重要性

计算机网络的应用对社会发展有着积极的作用,同时也必须注意它所带来的负面影响。计算机用户可通过网络快速地获取、传输与处理信息,这些信息涉及政治、经济、教育、科研与文化等领域。计算机网络在给广大用户带来方便的同时,也必然给别有用心的人带来可乘之机。例如,犯罪分子可以通过网络窃取商业机密、传播虚假信息、执行网络攻击等。网络用户也可能在无意中侵犯他人的隐私或者发表不恰当的言论。另外,计算机网络还可能引起知识产权、著作权等方面的纠纷。

计算机犯罪正在引起整个社会的普遍关注,而计算机网络已经成为犯罪分子攻击的重点。计算机犯罪是一种高技术型犯罪,其隐蔽性对网络安全构成了很大的威胁。有关统计资料表明,计算机犯罪案件以每年超过 100%的速度增长,网站被攻击的事件以每年 10 倍的速度增长。从 1986 年发现首例计算机病毒以来,病毒的类型和数量一直在逐年快速增长。根据国家计算机病毒应急处理中心发布的病毒预报,每周都有大量新病毒以及旧病毒的变种出现。攻击者在世界各地疯狂寻找攻击网络的机会,他们的活动几乎到了无孔不入的地步,政府网络与金融系统已经成为这些人的主要目标。

黑客(hacker)的出现是信息社会不容忽视的现象。黑客一度被认为是计算机狂热者的代名词,他们大多是对编程有着浓厚兴趣的大学生。后来,人们对黑客有了进一步的认识:黑客中的大部分人不伤害别人,但是也会做一些不该做的事情;部分黑客不顾法律与道德的约束,出于寻求刺激、被非法组织收买或出于报复心理,而肆意攻击与破坏一些企业、组织的

网络。近年来,黑客攻击的目的从最初的破坏网站、停止网络服务,转变为盗取用户密码、银行账号的有组织的经济犯罪。

互联网的广泛应用开始影响企业网的开发模式,用户希望在任何地方都可以方便地访问企业网中的计算机。但是,对于企业来说,将自己的企业网连入互联网也可能是一场噩梦。大多数的企业都有一些重要的内部资料,例如市场策略报告、客户名单、财务报表、产品开发计划等,这些经济情报的泄露对企业是致命的危险。如果企业网中的计算机遭到攻击,轻者会造成内部信息丢失或者被篡改,重者会造成整个管理信息系统的瘫痪,这些都会给企业造成严重的经济损失。

随着社会向高度信息化与网络化方向发展,社会对计算机网络越来越依赖,而网络安全问题也变得越来越严峻。如果我们将整个社会的人与人、人与物、物与物都连接在物联网环境中,那么网络安全将对国家安全产生深刻的影响。网络安全概念涉及的内容很广泛,它既包括用于解决网络应用中的安全威胁的各种技术或管理手段,也包括这些安全威胁本身以及相关活动。只有不断健全网络与信息安全方面的法律法规,提高管理人员的素质、法律意识与技术水平,提高用户遵守网络使用规则的自觉性,提高网络与信息系统安全防护水平,才能改善网络与信息系统的安全状况。

16.2.2 防火墙的基本概念

防火墙是网络安全技术的重要组成部分之一。防火墙(firewall)是在计算机网络之间执行控制策略的系统,它包括专用的硬件设备与软件系统。研究者在设计防火墙时所做的假设为:防火墙保护的内部网络是可信任的网络(trusted network),而其外部的网络是不可信任的网络(untrusted network)。图16-1给出了防火墙的基本结构。防火墙的目的是保护内部网络资源不被外部非授权用户使用,防止内部网络受到外部非法用户的攻击,因此防火墙的位置一定在内部网络与外部网络之间。

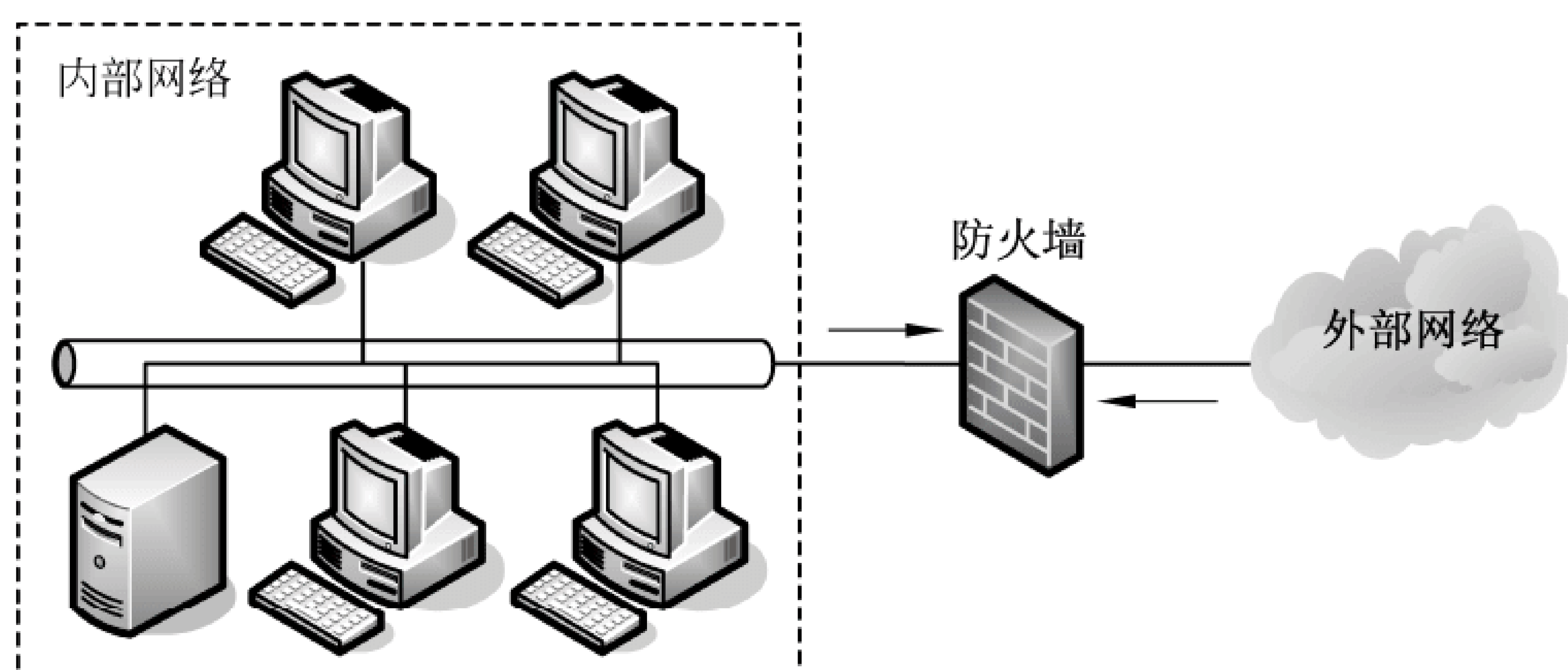


图16-1 防火墙的基本结构

防火墙在网络系统中扮演的角色是检查站(阻塞点或控制点)。防火墙的控制策略主要包括服务控制、方向控制、用户控制与行为控制。其中,服务控制是指确定访问的网络服务,例如FTP、电子邮件、Web服务等;方向控制是指确定访问网络服务的方向,例如从外部网络到内部或者从内部网络到外部;用户控制是指确定访问网络服务的用户,该策略主要应用于内部网络中的用户,对外部用户需使用安全鉴别技术,例如IPSec提供的认证服务;行为控制是指确定访问网络服务的形式,例如外部用户只能访问Web服务器的部分信息。

防火墙主要包括以下功能。

- 防火墙为内部网络提供安全入口,限制外部用户访问内部的特定资源。
- 防火墙为内部网络提供安全出口,限制内部用户访问外部的特定服务。
- 防火墙检查出入内部网络的数据包,过滤不安全的服务与非法的用户。
- 防火墙记录对内部网络的访问日志,提供对可疑行为进行报警的能力。
- 防火墙自身具有一定的防攻击能力,以便保证防御设施自身的安全性。

16.2.3 防火墙的分类方法

根据体系结构与实现技术的不同,防火墙主要分为两种类型:包过滤路由器与应用级网关。这是针对防火墙的最常用的分类依据。

1. 包过滤路由器

包过滤路由器(packet filtering router)是在网络层实现的防火墙系统,它通常是一台具有数据包过滤能力的路由器。包过滤路由器检查的数据包是 IP 分组,具体工作是检查 IP 分组中的某些内容,并且根据某种规则对 IP 分组执行相应操作。包过滤路由器的核心技术是包过滤规则,这些规则被用于匹配数据包中的相应内容,以便决定该数据包是被允许还是被拒绝。如果包过滤路由器允许某个数据包通过,则根据路由协议将它转发给相应的主机;如果包过滤路由器拒绝某个数据包通过,则丢弃数据包并且通知相应的发送方。包过滤路由器建立在路由技术的基础上。

包过滤路由器又称为屏蔽路由器(screening router),它通常是受保护网络与外界之间的第一道防线。包过滤路由器可用于内部子网之间的访问控制,这时主要考虑控制用户对特定资源的访问,而不是来自外部的黑客攻击。图 16-2 给出了包过滤路由器的工作原理。包过滤路由器根据内部设置的包过滤规则,检查进入路由器的每个数据包的相关内容,决定该数据包是否转发以及如何转发。普通路由器只检查数据包的网络层头部,并不检查数据包的传输层头部。包过滤路由器除了检查数据包的网络层头部,还可能检查数据包的传输层头部,其中的端口号是判断数据包类型的依据。

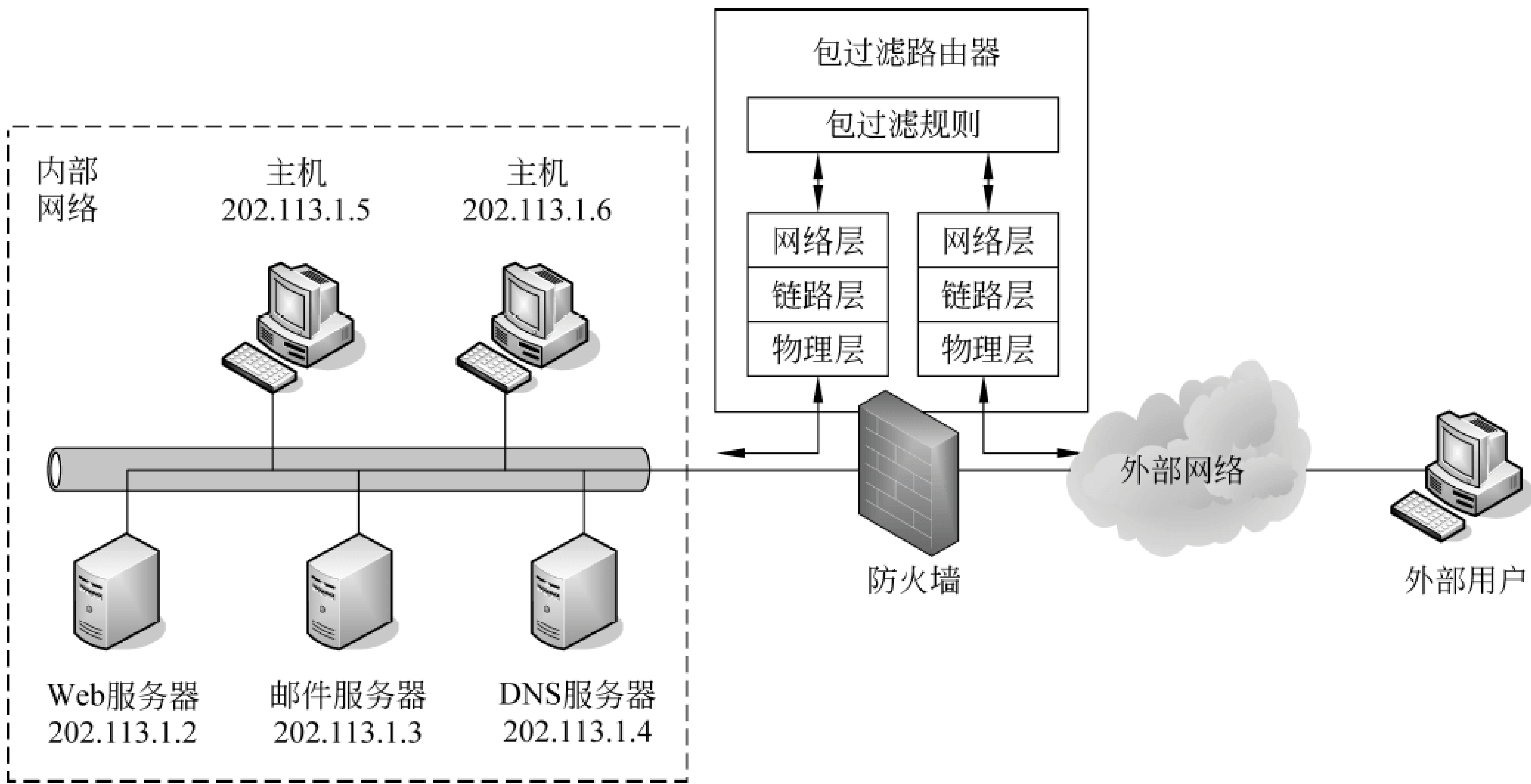


图 16-2 包过滤路由器的工作原理

包过滤路由器过滤的内容主要包括：①网络层的头部信息，例如IP地址、优先级(TOS)与协议类型(IP、ICMP、TCP、UDP等)；②传输层的头部信息，例如端口号(TCP与UDP)与TCP控制标记(SYN、ACK、FIN、RST等)。实现包过滤的关键是制定相应的包过滤规则。表16-1给出了包过滤规则的例子。其中，规则1表示任意主机发送给Web服务器的数据包都被转发；规则2表示任意主机发送给邮件服务器的数据包都被转发；规则3表示任意主机发送给DNS服务器的数据包都被转发；规则4说明任意主机发送给除Web服务器、邮件服务器、DNS服务器之外其他主机的数据包都被丢弃。

表16-1　包过滤规则的例子

规　则	源　地　址	目 的 地 址	传输层协议	端　口　号	操　作
1	任意	202.113.1.2	TCP	80	允许
2	任意	202.113.1.3	TCP	25	允许
3	任意	202.113.1.4	UDP	53	允许
4	任意	任意其他地址	任意	任意	丢弃

包过滤路由器的主要优点是：结构简单、便于管理、造价低廉。由于包过滤针对的是网络层与传输层的数据，因此这种操作对应用层是透明的，不需要客户机与服务器程序做任何修改。包过滤路由器的主要缺点是：包过滤规则配置比较困难，需熟悉各种协议及相关特征；包过滤建立在IP地址或端口号的基础上，只能控制到主机级而不能达到用户级；包过滤不能阻止某些类型的网络攻击，例如DDoS攻击与IP欺骗攻击。如果包过滤防火墙允许外部用户访问内部的Web服务器，它不关心通过80端口进入的数据包内容，这样就会为网络攻击提供可乘之机。

2. 应用级网关

应用级网关(application gateway)是在应用层实现的防火墙，通常是一台具有应用程序访问控制功能的主机。应用级网关处理的数据包是应用层数据，具体工作是检查数据包中的某些内容，并且根据某种规则对数据包执行相应的操作。应用级网关的核心技术是应用访问控制规则，这些规则被用于匹配数据包中的相应内容，以决定该数据包是被允许还是被拒绝。如果应用级网关允许某个数据包通过，则将该数据包转发给相应主机；如果应用级网关拒绝某个数据包通过，则丢弃数据包并通知相应的发送方。

应用级网关是建立在代理技术的基础上。应用级网关又称为应用级代理(application proxy)，它通常是受保护网络与外界之间的第二道防线。应用级代理实现身份认证与服务请求合法性检查。由于应用级网关实现对应用程序的访问控制，因此其控制与过滤功能通常是通过软件实现的。图16-3给出了应用级网关的工作原理。应用级网关可以同时支持有限的多种应用，例如Web、电子邮件、DNS等。应用级网关的主要功能是检查应用层数据，然后决定是否允许该数据包进入内部网络，它实现的是用户级而不是主机级认证。

16.2.4　防火墙系统结构

防火墙通常由包过滤路由器与应用级网关作为基本单元，采用多级结构与多种组合方式而形成的系统。这里，包过滤路由器通常用字符S来表示；堡垒主机是指一台运行应用级网关软件的主机，通常用字符B来表示。堡垒主机又可分为两种：单接口堡垒主机与双接

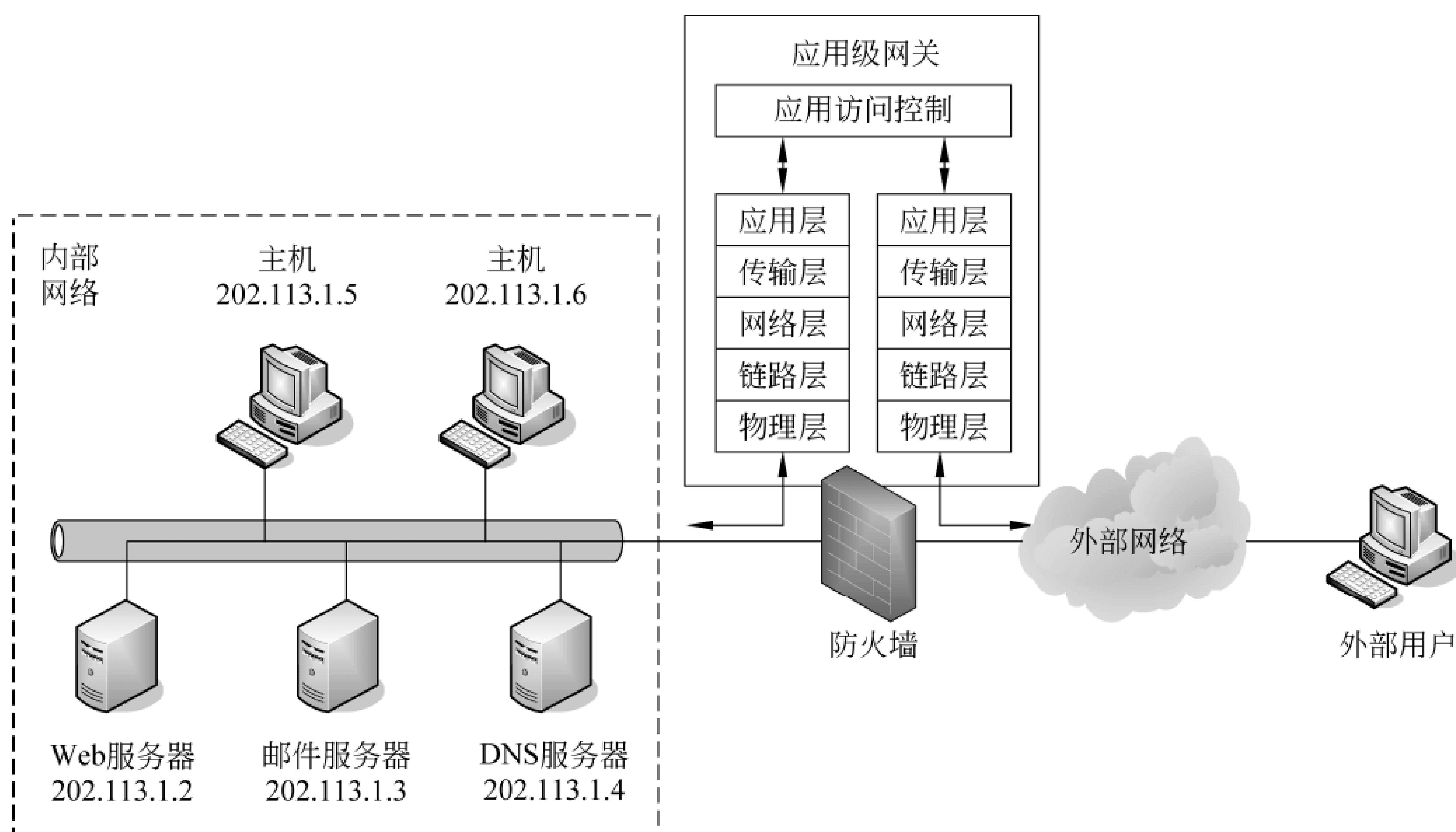


图 16-3　应用级网关的工作原理

口堡垒主机。其中,单接口堡垒主机有一个网络接口,可连接一个子网,通常用字符 B1 来表示;双接口堡垒主机有两个网络接口,可连接两个子网,通常用字符 B2 来表示。因此,防火墙可用 S 与 B 组合来表示,例如 S-B1。

根据主机面向的服务对象的差异,内部网络主机可分为三种类型:普通主机(如内部用户的工作主机)、对内服务器(如文件服务器、数据库服务器)、对外服务器(如 Web 服务器、FTP 服务器)。普通主机与对内服务器仅面对内部用户,通常不允许外部用户对它们进行访问;对外服务器可以面对内部用户与外部用户,通常允许外部用户来访问它们,但是需要为用户设置相应的访问权限。因此,研究者提出建立非军事区(De Militarized Zone, DMZ),将可对外服务器放置在这个区域中,以便为内部网络提供一定安全缓冲。图 16-4 给出了非军事区的基本结构。

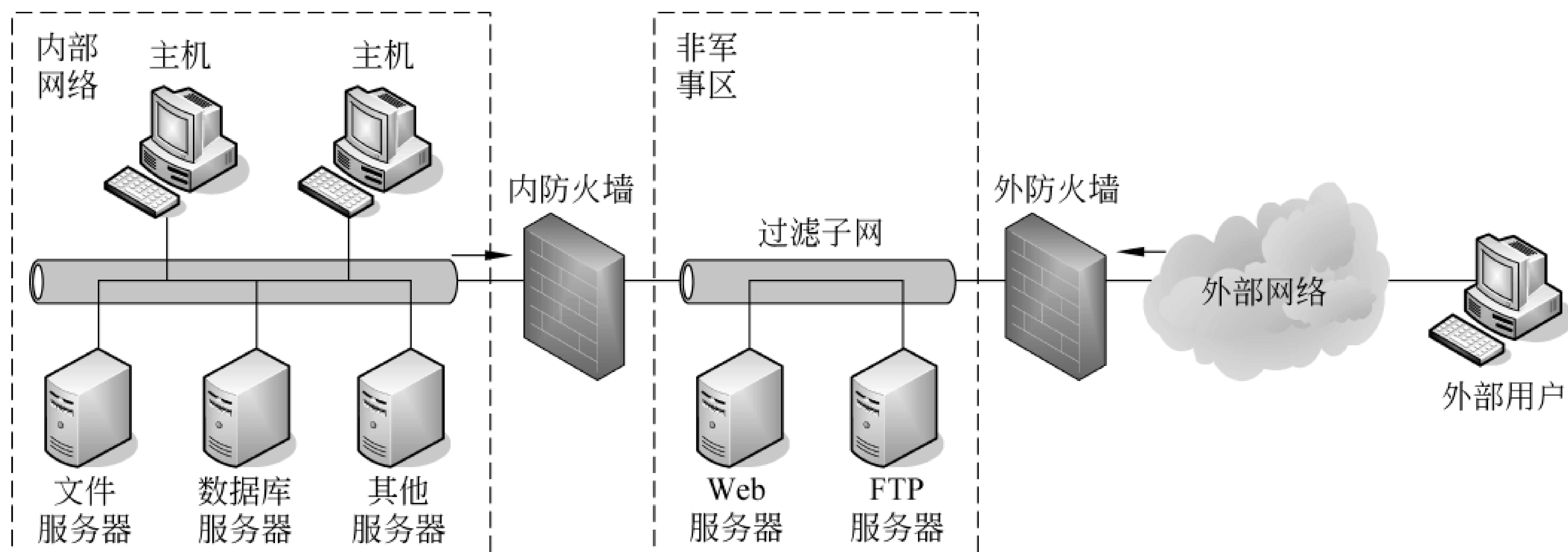


图 16-4　非军事区的基本结构

在实际的网络系统中,不同系统在网络规模、拓扑结构等方面有较大的差异,对网络资

源的防护目标与安全策略也有一定的差别,显然不同系统中使用的防火墙不可能相同。根据网络系统是否对外提供服务以及服务类型等因素,在网络系统中可以选择是否设置非军事区,以及设置一个还是多个非军事区。防火墙系统通常由包过滤路由器与堡垒主机共同构成,它可能采用的体系结构有多种组合方式。典型的防火墙系统结构主要包括S-B1、S-B2、S-B1-S-B1、S-B2-B2等。

16.3 例题分析

16.3.1 设计要求

根据包过滤路由器的工作原理,编写包过滤程序,捕获与分析网络中的IP包,并且将符合条件的IP包信息显示在控制台上。在本练习中为了简便起见,只设置两条简单的包过滤规则,并且不丢弃符合条件的IP包。两条包过滤规则分别是:目的地址为202.113.16.10、协议类型为UDP、允许通过;源地址为202.113.16.10、协议类型为UDP、拒绝通过。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为PackFilter.exe,则程序的命令行格式为:

```
PackFilter packet_sum
```

其中,packet_sum为符合包过滤规则的IP包数量。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
--------------------------------------
源 IP 地址:xx.xx.xx.xx
目的 IP 地址:xx.xx.xx.xx
协议类型:UDP
操作类型:允许或拒绝
--------------------------------------
...
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

16.3.2 关键问题

1. 初始化Socket结构

为了通过网卡截获网络中传输的IP包,需要使用socket()函数创建原始套接字。首先,调用setsockopt()函数自行处理IP包头部,将第三个参数设置为IP_HDRINCL,并将flag设置为true。接着,对原始套接字地址进行填充,其中的IP地址应填写本机IP地址,可以通过gethostname()与gethostbyname()函数来获得,端口号可在规定的范围中随意填

写。最后,使用 bind()函数将套接字与网卡绑定。

下面给出初始化 Socket 结构的伪代码:

```
//套接字异步启动
WSAStartup(MAKEWORD(2,2),&WSAData);
//创建原始 Socket
socket(AF_INET,SOCK_RAW,IPPROTO_IP);
//设置 IP 头操作选项
BOOL flag=true;
setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char*)&flag,sizeof(flag));
//获得本地主机名
gethostname(hostName,100);
//获取本地 IP 地址
gethostbyname(hostName);
//填充 Socket 地址
sockaddr_in host_addr;
host_addr.sin_family=AF_INET;
host_addr.sin_port=htons(6000);
host_addr.sin_addr=*(in_addr*)pHostIP->h_addr_list[0];
//socket 绑定本地网卡
bind(sock,(PSOCKADDR)&host_addr,sizeof(host_addr));
```

2. 接收所有 IP 包

在通常情况下,网卡不会接收目的地址不是自己的 IP 包。如果想截获经过网卡的所有 IP 包,需要调用 WSAIoctl()函数将网卡设置为混杂模式,当接收 IP 包中的协议类型与原始套接字匹配,IP 包内容就被复制到套接字缓冲区中。然后,可以循环调用 recv()函数来接收所有 IP 包。

下面给出接收经过网卡 IP 包的伪代码:

```
//网卡设置为混杂模式
DWORD dwBufferLen[10];
DWORD dwBufferInLen=1;
DWORD dwBytesReturned=0;
WSAIoctl(SnifferSocket,IO_RCVALL,&dwBufferInLen,sizeof(dwBufferInLen),
&dwBufferLen,sizeof(dwBufferLen),&dwBytesReturned,NULL,NULL);
//开始接收所有 IP 包
while(i<packsum)
    recv(sock,buffer,65535,0);
```

3. 检查包过滤规则

在成功接收经过网卡的每个 IP 包后,需要根据包过滤规则分析 IP 头部字段。首先,利用结构体来定义包过滤规则结构,并且按照需要填充好每条包过滤规则内容。然后,根据每条规则依次检查 IP 包头部相关字段,如果符合规则就对 IP 包执行相应操作。

下面给出检查包过滤规则的伪代码:

```
//定义包过滤规则表
typedef struct
{
    char SourceAddr[16];
    char DestinAddr[16];
    unsigned short SourcePort;
    unsigned short DestinPort;
    unsigned char Protocol;
    bool Operation;
}filter_table;
//填写包过滤规则
filter_table filter[2];
memcpy(filter[0].SourceAddr,"202.113.16.10",strlen("202.113.16.10"));
filter[0].Protocol=UDP_PRO;
filter[0].Operation=REJECT_OPT;
//检查包过滤规则
if(strcmp(source_ip,filter[0].SourceAddr)==0)
    if(ip.Protocol==filter[0].Protocol)
        …
```

4. 程序流程图

图 16-5 给出了主程序流程图。这里,要求输入的命令行参数必须正确,除了程序本身的名称以外,还需要有一个符合过滤规则的包个数。如果命令行参数的个数不是一个,则程序在输出错误信息后退出。在主程序的流程中,需要判断是否捕获 IP 包以及是否符合包过滤规则。

16.3.3 程序源代码

下面给出包过滤程序的源代码:

```
//PackFilter.cpp : 定义控制台应用程序的入口点

#include "stdafx.h"
#include "string.h"
#include "winsock2.h"
#include "ws2tcpip.h"
#include "iostream"
using namespace std;

#pragma comment(lib,"ws2_32")                        //加载 ws2_32.lib

#define IO_RCVALL _WSAIOW(IOC_VENDOR,1)
#define TCP_PRO 6                                    //定义过滤协议
#define UDP_PRO 17
#define PERMIT_OPT 0                                 //定义过滤操作
```

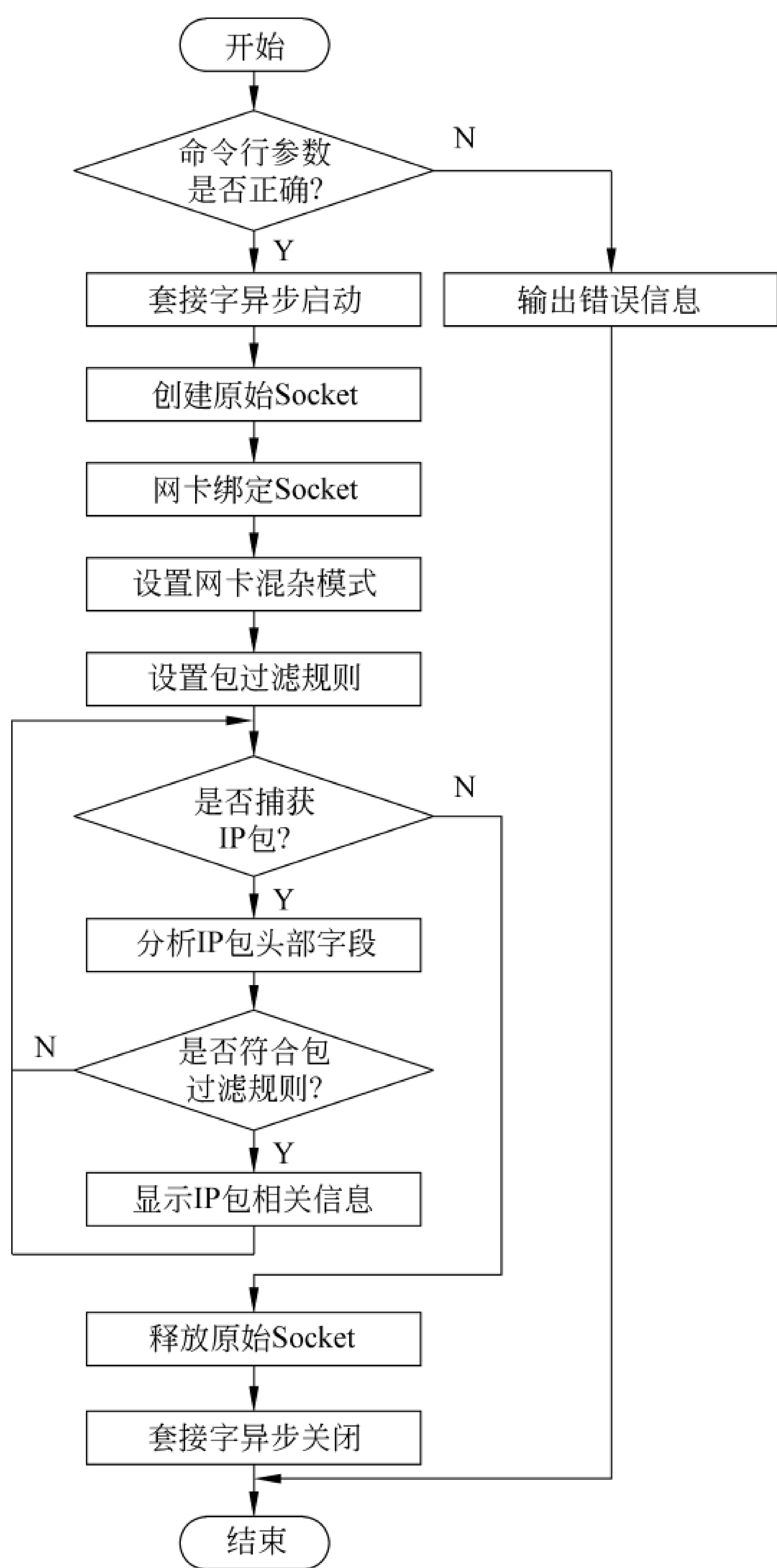


图 16-5　主程序流程图

```
#define REJECT_OPT 1

typedef struct                                   //定义 IP 头部结构
{
    union
    {
        unsigned char Version;                   //版本 (字节前 4 位)
        unsigned char HeadLen;                   //头部长度 (字节后 4 位)
    };
    unsigned char ServiceType;                   //服务类型
    unsigned short TotalLen;                     //总长度
```

```
        unsigned short Identifier;                          //标识符
        union
        {
            unsigned short Flags;                           //标志位(字前3位)
            unsigned short FragOffset;                      //片偏移(字后13位)
        };
        unsigned char TimeToLive;                           //生存周期
        unsigned char Protocol;                             //协议
        unsigned short HeadChecksum;                        //头部校验和
        unsigned int SourceAddr;                            //源 IP 地址
        unsigned int DestinAddr;                            //目的 IP 地址
        unsigned char Options;                              //选项
    }ip_head;

    typedef struct                                          //定义包过滤规则表
    {
        char SourceAddr[16];                                //源 IP 地址
        char DestinAddr[16];                                //目的 IP 地址
        unsigned short SourcePort;                          //源端口号
        unsigned short DestinPort;                          //目的端口号
        unsigned char Protocol;                             //协议类型
        bool Operation;                                     //操作类型
    }filter_table;

    void main(int argc, char* argv[])
    {
        if(argc!=2)                                         //检查命令行参数
        {
            cout<<endl<<"请按以下格式输入命令行:PackFilter packet_sum"<<endl;
            return;
        }

        WSADATA WSAData;                                    //套接字异步启动
        if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0)
        {
            cout<<endl<<"WSAStartup 初始化失败"<<endl;
            return;
        }

        SOCKET sock=socket(AF_INET,SOCK_RAW,IPPROTO_IP);
        if(sock==INVALID_SOCKET)                            //创建原始 Socket
        {
            cout<<endl<<"创建 Socket 失败!"<<endl;
            return;
```

```
}

BOOL flag=true;                                         //设置 IP 头操作选项
if(setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char*)&flag,sizeof(flag))==SOCKET_ERROR)
{
    cout<<endl<<"setsockopt 操作失败"<<endl;
    return;
}

char hostName[128];
if(gethostname(hostName,100)==SOCKET_ERROR)             //获得本地主机名
{
    cout<<endl<<"gethostname 操作失败"<<endl;
    return;
}

hostent* pHostIP;
if((pHostIP=gethostbyname(hostName))==NULL)             //获取本地 IP 地址
{
    cout<<endl<<"gethostbyname 操作失败"<<endl;
    return;
}

sockaddr_in host_addr;                                  //填充 Socket 地址
host_addr.sin_family=AF_INET;
host_addr.sin_port=htons(6000);
host_addr.sin_addr=*(in_addr*)pHostIP->h_addr_list[0];

if(bind(sock,(PSOCKADDR)&host_addr,sizeof(host_addr))==SOCKET_ERROR)
{                                                       //socket 绑定本地网卡
    cout<<endl<<"bind 操作失败"<<endl;
    return;
}

DWORD dwBufferLen[10];
DWORD dwBufferInLen=1;
DWORD dwBytesReturned=0;
if(WSAIoctl(sock,IO_RCVALL,&dwBufferInLen,sizeof(dwBufferInLen),
&dwBufferLen,sizeof(dwBufferLen),&dwBytesReturned,NULL,NULL)==SOCKET_
ERROR)
{                                                       //设置 socket 接收所有包
    cout<<endl<<"WSAIoctl 操作失败"<<endl;
    return;
```

```
    }

    filter_table filter[2];                          //填写包过滤规则(2项)
    memset(filter[0].SourceAddr,0,16);
    memcpy(filter[0].SourceAddr,"202.113.16.10",strlen("202.113.16.10"));
    filter[0].Protocol=UDP_PRO;
    filter[0].Operation=REJECT_OPT;
    memset(filter[1].DestinAddr,0,16);
    memcpy(filter[1].DestinAddr,"202.113.16.10",strlen("202.113.16.10"));
    filter[1].Protocol=UDP_PRO;
    filter[1].Operation=PERMIT_OPT;

    cout<<endl<<"开始过滤 IP 包:"<<endl;
    char buffer[65535];                              //设置接收缓冲区
    int packsum=atoi(argv[1]);
    int i=0;
    while(i<packsum)                                 //开始接收 IP 包
    {
        if(recv(sock,buffer,65535,0)>0)
        {
            ip_head ip=*(ip_head*)buffer;            //获得 IP 包头部

            int addrlen;
            char source_ip[16];                      //解析源 IP 地址
            memset(source_ip,0,16);
            addrlen=strlen(inet_ntoa(*(in_addr*)&ip.SourceAddr));
            memcpy(source_ip,inet_ntoa(*(in_addr*)&ip.SourceAddr),addrlen);

            char destin_ip[16];                      //解析目的 IP 地址
            memset(destin_ip,0,16);
            addrlen=strlen(inet_ntoa(*(in_addr*)&ip.DestinAddr));
            memcpy(destin_ip,inet_ntoa(*(in_addr*)&ip.DestinAddr),addrlen);

            //检验包过滤规则 1
            if(strcmp(source_ip,filter[0].SourceAddr)==0)
            {
                if(ip.Protocol==filter[0].Protocol)
                {
                    cout<<"--------------------------------------------"
                    <<endl;
                    cout<<"源 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.SourceAddr)<<
                   endl;
                    cout<<"目的 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.DestinAddr)<<
                    endl;
                    cout<<"协议类型:UDP"<<endl;
```

```
                    cout<<"操作类型:拒绝"<<endl;
                    i++;
                }
            }

            //检验包过滤规则 2
            if(strcmp(destin_ip,filter[1].DestinAddr)==0)
            {
                if(ip.Protocol==filter[1].Protocol)
                {
                    cout<<"----------------------------------------"
                    <<endl;
                    cout<<"源 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.SourceAddr)<<
                   endl;
                    cout<<"目的 IP 地址:"<<inet_ntoa(*(in_addr*)&ip.DestinAddr)<<
                    endl;
                    cout<<"协议类型:UDP"<<endl;
                    cout<<"操作类型:允许"<<endl;
                    i++;
                }
            }
        }
    }

    closesocket(sock);                              //关闭原始 Socket
    WSACleanup();                                   //套接字异步关闭
}
```

图 16-6 给出了包过滤程序的执行过程。程序命令行输入为 PackFilter 3。包过滤程序首先以混杂模式截获经过的三个 IP 包,然后根据两条过滤规则依次解析每个 IP 包的相关字段,并且将符合条件的 IP 包信息显示在控制台上。

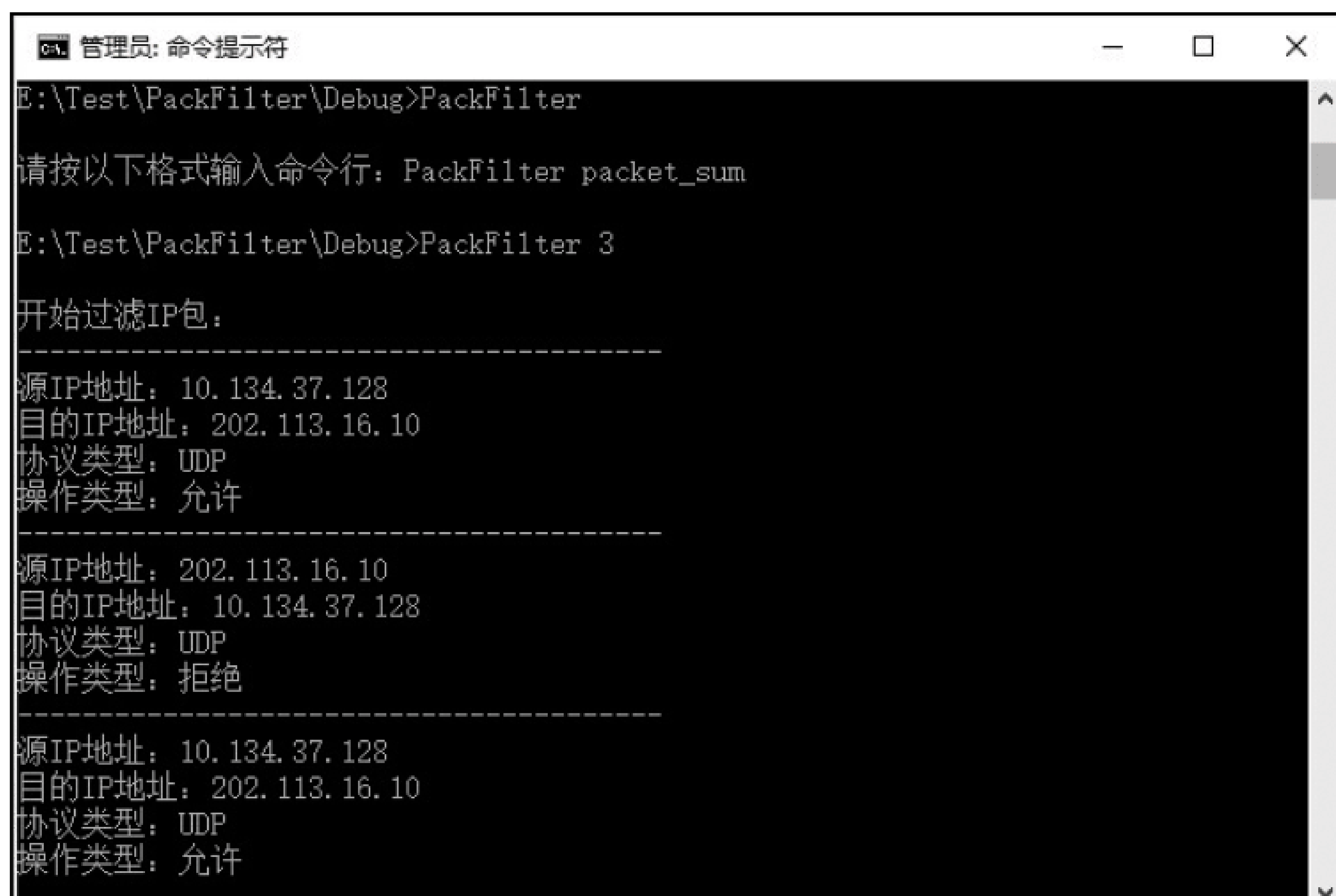


图 16-6　包过滤程序的执行过程

16.4 练　习　题

根据包过滤路由器的工作原理,编写包过滤程序,捕获与分析网络中的IP包,并且将符合条件IP包信息显示在控制台上。由学生自己设计复杂一些的包过滤规则。在本练习中为了简便起见,不需要对符合条件的IP包进行丢弃。程序设计的具体要求如下。

(1) 要求程序为命令行程序。例如,可执行文件名为PackFilter.exe,则程序的命令行格式为:

```
PackFilter packet_sum
```

其中,packet_sum为符合包过滤规则的IP包数量。

(2) 要求将部分字段内容显示在控制台上,具体格式为:

```
------------------------------------
源 IP 地址:xx.xx.xx.xx
目的 IP 地址:xx.xx.xx.xx
协议类型:TCP 或 UDP
操作类型:允许或拒绝
------------------------------------
...
```

(3) 要求有良好的编程规范与注释。编程所使用的操作系统、语言和编译环境不限,但是在提交的说明文档中需要加以注明。

(4) 要求撰写说明文档,包括程序的开发思路、工作流程、关键问题、解决思路以及进一步的改进等内容。

附录

RFC 文档

RFC 文档可以从 RFC 的官方网站(http//www.rfc-editor.org/rfc.html)或其他镜像网站得到。在进行网络软件编程时,可能会用到以下这些 RFC 文档。

RFC768 User Datagram Protocol,J. Postel,August 1980

RFC791 Internet Protocol,J. Postel,September 1981

RFC792 Internet Control Message Protocol,J. Postel,September 1981

RFC793 Transmission Control Protocol,J. Postel,September 1981

RFC826 Ethernet Address Resolution Protocol,D. C. Plummer,November 1982

RFC854 Telnet Protocol Specification,J. Postel,J. Reynolds,May 1983

RFC950 Internet Standard Subnetting Procedure, J. C. Mogul, J. Postel, August 1985

RFC959 File Transfer Protocol,J. Postel,J. Reynolds,October 1985

RFC1034 Domain Names-Concepts and Facilities,P. Mockapetris,November 1987

RFC1035 Domain Names-Implementation and Specification, P. Mockapetris, November 1987

RFC1157 Simple Network Management Protocol,J. Case,M. Fedor,M. Schoffstall, May 1990

RFC1349 Type of Service in the Internet Protocol Suite,P. Almquist,July 1992

RFC1700 Assigned Numbers,J. Reynolds,J. Postel,October 1994

RFC1939 Post Office Protocol Version 3,J. Myers,M. Rose,May 1996

RFC1945 Hypertext Transfer Protocol-HTTP/1.0, T. B. Lee, R. Fielding, H. Frystyk, May 1996

RFC2401 Security Architecture for The Internet Protocol, S. Kent, R. Atkinson, November 1998

RFC2402 IP Authentication Header,S. Kent,R. Atkinson,November 1998

RFC2406 IP Encapsulating Security Payload (ESP), S. Kent, R. Atkinson, November 1998

RFC2460 Internet Protocol Version 6 (IPv6) Specification, S. Deering, R. Hinder, December 1998

RFC2463　Internet Control Message Protocol(ICMPv6) for the Internet Protocol Version 6(IPv6) Specification, A. Conta, S. Deering, December 2006

RFC2581　TCP Congestion Control, M. Alman, V. Panon, W. Stevens, April 1999

RFC2821　Simple Mail Transfer Protocol, J. Klensin, April 2001

RFC3493　Basic Socket Interface Extensions for IPv6, R. Gilligan, S. Thomson, J. Bound, February 2003

RFC3513　Internet Protocol Version 6(IPv6) Addressing Architecture, R. Hinder, S. Deering, April 2003

RFC3542　Advanced Socket Application Program Interface (API) for IPv6, W. Stevens, M. Thomas, E. Nordmark, May 2003

参考文献

[1] Andrew S Tanenbaum.计算机网络.英文版.5版.北京：机械工业出版社，2011.

[2] 吴功宜，吴英.计算机网络高级教程.2版.北京：清华大学出版社，2015.

[3] 吴功宜，董大凡，王珺，等.计算机网络高级软件编程技术.2版.北京：清华大学出版社，2011.

[4] 吴功宜，张健，董大凡，等.网络安全高级软件编程技术.北京：清华大学出版社，2007.

[5] 吴功宜.计算机网络与互联网技术研究、应用和产业发展.北京：清华大学出版社，2008.

[6] 吴功宜，吴英.计算机网络技术教程：自顶向下分析与设计方法.北京：机械工业出版社，2009.

[7] 吴功宜，吴英.计算机网络课程设计.2版.北京：机械工业出版社，2015.

[8] 凯文 R.福尔，W.理查德·史蒂文斯.TCP/IP详解 卷1：协议.吴英，张玉，许昱玮，译.北京：机械工业出版社，2016.

[9] 福罗赞.TCP/IP协议族.4版.王海，张娟，朱晓阳，译.北京：清华大学出版社，2011.